Porting Guide
for RADVISION Protocol Stacks

RADVISION
Delivering the Visual Experience

NOTICE

For further information contact RADVISION or your local distributor or reseller.

Porting Guide for RADVISION Protocol Stacks, January, 2006
Publication 7
http://www.radvision.com

CONTENTS

4 *Threading Modules*

5 *Networking Modules*

About this Manual

What's in the Porting Guide?

The *Porting Guide for RADVISION Protocol Stacks* is for developers of RADVISION Stack applications who wish to connect the Stacks to their operating systems using Low Level Porting Functions. This *Porting Guide* provides an introduction to the Low Level Porting Functions, lists the functions and provides each function with a general description, syntax description and other important information. The *Porting Guide* also contains constants, enumerations and type definitions.

1

OVERVIEW

INTRODUCTION

RADVISION Stacks are platform/operating system (OS) independent thanks to their modular structure, which consists of two integrated parts:

- The upper layer—the pure implementation of the Stack (SIP, H.323, etc.)
- The lower layer—the Common Core library, which provides the upper layer with an OS-independent API for a large number of OS services

Figure 1-1 illustrates the relationship between the upper layer and the Common Core.

The upper layer uses the Common Core API to access OS services transparently, without specific knowledge of the OS architecture and its API underneath. In other words, the Common Core is the bridge between RADVISION Stacks and the operating system.

RADVISION Stacks provide full implementation of the Common Core for a variety of operating systems. See Table 1-1 for the most up-to-date list of supported operating systems.

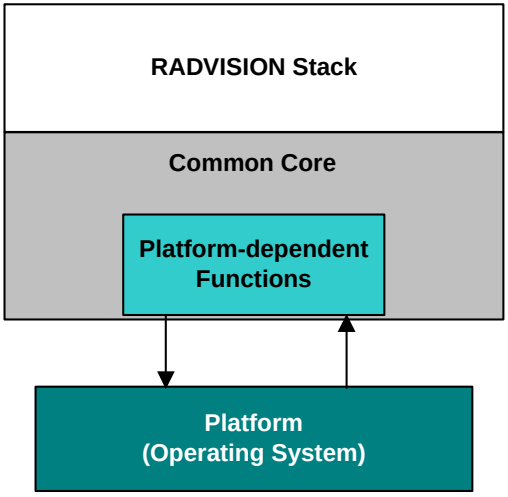

Figure 1-1 *Position of Low Level (Porting) Functions*

You can implement the platform-dependent functions by referring to the provided source code. The source code provides an example of how to write platform-dependent functions for your specific operating system.

You do not need to implement all of the platform-dependent functions described in this Porting Guide for a specific RADVISION Stack to be compatible with the operating system. The platform-dependent functions do not need to be implemented for the operating systems listed above, since out-of-the-box compatibility for these operating systems is provided.

Table 1-1 *List of Supported Operating Systems*

OS	Version
Solaris	2.8
	2.9
	2.10
Linux RedHat	9.0
	AS
	WS
	ES
Linux SUSE	8
	9
Linux MontaVista	3.0
	3.1
MAC OS	8
FreeBSD	5.4
VxWorks	5.4
	5.5
	6.1
PSOS	3.0
Win. CE	4.2
	5.0

Table 1-1 *List of Supported Operating Systems*

Windows	XP
	2000 Pro.
	2003
NUCLEUS	4.4

Note The list of supported operating systems refers only to the Common Core itself. The actual Stack that you are using might support fewer operating systems. Not all of the operating systems are distributed together in the same package.

COMMON CORE

The Common Core is an independent RADVISION project holding all modules that are related to the operating system and to the common functionality needed by different Stacks. It includes:

- The operating system-specific code (referred to as the Low Level Porting Functions).
- Various general types that are used by the Stacks, such as RvUint32, RvBool, RvStatus, and so on.
- Basic modules that are not operating system-specific, but have common uses of the Stacks, such as timers, message queues, and so on.
- Utility functions needed by the modules of the Common Core itself.

The modules of the Common Core are internal to the Stacks and applications should not access them directly, since their interfaces may largely vary between versions.

Not all of the Common Core needs to be ported—only the Low Level Porting Functions, which are described in this document.

ADAPTERS

Some of the modules in the Common Core are by nature platform dependent. These modules consist of two layers:

- The OS-independent layer, which contains the universal part of the module's implementation. This part is suitable for all of the supported operating systems.

- The OS-dependent layer, which contains the OS-specific part of the module's implementation. This part should be tailored individually for each OS.

The implementation (some of the) platform-dependent modules in the Common Core is now based on the Adapter concept. According to this concept, each module implements only the universal part of the code, while the OS-dependent part is implemented as a collection of functions—located in a separated file: the Adapter—that are called by the generic part.

The Adapters are organized in folders that contain several pairs of source files. Each pair of source files represents a single Adapter and includes one C file and one H file. The C file contains the OS-dependent functions, which are called by the OS-independent layer. The H file defines the OS-dependent structures and typedef's.

The Adapters are distributed among the folders in a way that allows you to choose a predefined set of folders that includes all of the required Adapters for a specific operating system. The make process will compile all of the Adapters in the selected folders.

For each OS, only a subset of the Adapters is compiled and linked with the Common Core. The specific subset is determined by the relevant make file located in the *"common/config/make"* directory, and is specified in the variable:

`ADAPTERS_$(TARGET_OS).`

For example, for Red Hat Linux, in the file *"common/config/make/os_linux.mak"* the required Adapters are defined as follows:

```
ADAPTERS_$(TARGET_OS) := pmutex posix linux
```

This means that only the Adapters in the **pmutex**, **posix**, and **linux** folders will be compiled and linked with the Common Core library.

PORTING MODULES

Listed below are the Low Level modules and their sub-modules, each with its own set of files:

- **Memory Handling Module**
  - OsMem —standard dynamic memory management using standard operating system calls.
- **Timing Modules**
  - Timestamp—functions for accessing a timestamp. Timestamp values are guaranteed to be linear (will never go backwards) and will effectively never wrap.
  - Clock—functions for accessing the wall clock (time of day).

  - Tm—functions for creating and manipulating calendar times. This is basically the same as the ANSI tm structure with the addition of nanoseconds, accessor functions, and more natural ranges for some parameters.

- **Threading Modules**
  - Semaphore—semaphore functions.
  - Lock—non-recursive locking functions to use specifically for locking code sections.
  - Mutex—recursive locking functions to use specifically for locking code sections.
  - Thread—functions for creating and manipulating threads.
- **Networking Modules**
  - Socket—functions for handling socket operations.
  - Select—select() like interface functions used to check sockets for events.
  - Host—functions for accessing the current name and addresses of the host.
  - TLS (Transport Layer Security)—functions for managing TLS sessions, sending and receiving data via these TLS sessions.
- **Utilities and Standard ANSI Modules**
  - 64Ascii—functions for converting 64 bit numbers to strings.
  - Assert—replacement of the ANSI assert.h for portability.
  - StdIo—replacement of the ANSI stdio.h for portability.
  - LogListener—functions for log listeners used by the various Stacks.

An additional set of operating system-related type definitions, makefiles and compiler settings should be ported.

PORTING THE MODULES

When porting each of the modules, please note the following:

- Each module is written as a set of interface implementations. For example, the Select module implements WinSock, select(), poll() and other interfaces independently. Only one interface is then compiled, using the compiler definitions. You should find the existing interface that is closest to your operating system of choice and then try to use that interface, or write your own interface as part of the code itself.
- Each module has two initialization functions, Rv<module>Init() and Rv<module>End(). These functions are for initializing and destroying global information. An example of where these functions can be used is the Select module, where WSAStartup() must be called before any of the WinSock functions can be used in the Windows operating system. You should use these functions to initialize any global information you need, or just leave their implementation empty if they are not needed. These functions will not be documented in each specific module in this Porting Guide.
- Rv<module>Init() is guaranteed to be called only once during Stack operation, at the beginning of execution. In the same way, Rv<module>End() is guaranteed to be called once when the Stacks are destructed. Rv<module>Init and Rv<module>End functions should be executed from the same thread, and it is the responsibility of the user to ensure that no other core functions will be applied before Rv<module>Init.
- Most of the module functions return an RvStatus parameter. RvStatus is defined as a 32-bit integer. In case of success, the value of the parameter is RV_OK (=0). In case of failure, the value of the parameter is negative, while warnings (which might not be failures) are indicated as positive values. In case of error or warning, the returned RvStatus consists of the following:
  - Error or warning code—Ten less significant bits
  - Module code—Ten bits that are more significant than the error or warning code bits
  - Library code—Ten bits that are more significant than the module code bits

- Many of the module functions receive a pointer to the log manager instance structure as an input parameter. The log manager instance parameter is used to print core layer log messages. When porting core functions, please note that the log manager instance parameter might be NULL.

Porting by Protocol Stack

Please see the section Porting by Protocol Stack in the Programmer Guide of the relevant RADVISON Protocol Stack for a list of Common Core modules to be ported, since not all of the modules are used by each Stack.

2

TIMING MODULES

All of the Stacks need timing functions. The functionality can be split into two significant parts:

- A way of knowing the current “wall clock” (actual time).
- A way of knowing how much time has passed (getting a “timestamp” and finding the time difference between a timestamp and earlier ones).

The functions, constants and type definitions included in this chapter are:

TIMESTAMP MODULE

Timestamp Functions

- RvTimestampGet()
- RvTimestampResolution()

CLOCK MODULE

Clock Constants and Type Definitions

- RvTime

Clock Functions

- RvClockGet()
- RvClockResolution()
- RvClockSet()

TM MODULE

Tm Constants and Type Definitions

- RvTm
- RV_TM_xxx

Tm Functions

- RvTmConvertToTime()
- RvTmConstructUtc()
- RvTmConstructLocal()
- RvTmAsctime()
- RvTmStrftime()

TIMESTAMP MODULE

The Timestamp module provides functions for accessing a timestamp. Timestamp values are guaranteed to be linear (will never go backwards) and will effectively never wrap.

You can find the module in *ccore/rvtimestamp.c*

ASSUMPTIONS YOU NEED TO FOLLOW

- Timestamps are given in nanoseconds. On operating systems that do not support this granularity, you should multiply the resolution you have to achieve the semblance of a nanoseconds resolution (as done by some of the interfaces already implemented in this module).
- Timestamps should not wrap while the Stacks are running. Since timestamps are 64 bits long, there should be no problem in achieving this goal.

TIMESTAMP FUNCTIONS

RvTimestampGet()

DESCRIPTION

Gets a timestamp value in nanoseconds.

Values returned by subsequent calls are guaranteed to be linear (will never go backwards) and will never wrap.

SYNTAX

```
RvInt64 RvTimestampGet(
  IN RvLogMgr *logMgr);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns the nanasecond timestamp.

RvTimestampResolution()

DESCRIPTION

Gets the resolution of the timestamp in nanoseconds.

SYNTAX

```
RvInt64 RvTimestampResolution(
  IN RvLogMgr *logMgr);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns the resolution of the timestamp in nanoseconds.

REMARKS

Check the Stack you are using to see if it uses this function—you may be able to leave the implementation of this function empty.

CLOCK MODULE

The Clock module provides functions for accessing the wall clock (time of day). Some of the Stacks need this module for logging purposes only.

You can find the module in *ccore/rvclock.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- None.

Clock Constants and Type Definitions

RvTime

A time value in seconds and nanoseconds.

Syntax

```
typedef struct
{
  RvInt32 sec;
  RvInt32 nsec;
}RvTime;
```

CLOCK FUNCTIONS

RvClockGet()

DESCRIPTION

Gets time of day in seconds and nanoseconds since epoch (January 1, 1970). The epoch can vary due to system time adjustments.

SYNTAX

```
RvInt32 RvClockGet(
  IN  RvLogMgr    *logMgr,
  OUT RvTime      *t);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

t

A pointer to time structure where the time will be stored. If NULL, the seconds will still be returned.

RETURN VALUES

Returns the number of seconds since January 1, 1970 and complements the result with seconds and nanoseconds.

RvClockResolution()

DESCRIPTION

Gets the resolution of the wall clock in seconds and nanoseconds.

SYNTAX

```
RvInt32 RvClockResolution(
  IN  RvLogMgr    *logMgr,
  OUT RvTime      *t);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

t

A pointer to time structure where resolution will be stored. If NULL, the nanseconds will still be returned.

RETURN VALUES

Returns the nanoseconds portion of the clock resolution.

REMARKS

Check the Stack you are using to see if it uses this function—you may be able to leave the implementation of this function empty.

RvClockSet()

DESCRIPTION

Sets the time of day in seconds and nanoseconds since January 1, 1970.

SYNTAX

```
RvStatus RvClockSet(
  IN const       RvTime *t,
  IN RvLogMgr    *logMgr);
```

PARAMETERS

t

A pointer to the time structure with the time to be set.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- Check the Stack you are using to see if it uses this function—you may be able to leave the implementation of this function empty.
- On some systems the nanoseconds portion of the time will be ignored.
- On some systems setting the clock requires special privileges.

TM MODULE

The Tm module provides functions for creating and manipulating calendar times. This module is basically the same as the ANSI tm structure with the addition of nanoseconds, accessor functions, and more natural ranges for some parameters. Some of the Stacks need this module for logging purposes only.

You can find this module in *ccore/rvtm.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- None.

TM CONSTANTS AND TYPE DEFINITIONS

RvTm

A calendar time object.

Syntax

```
typedef struct
{
  struct tm tm;/* use ANSI tm definition for simplicity */
  RvInt32 nsec;
}RvTm;
```

RV_TM_xxx

RV_TM_ASCTIME_BUFSIZE

The minimum size of character buffer required by RvTimeAsctime

```
#define RV_TM_ASCTIME_BUFSIZE 30
```

Daylight Savings Constants

Constants for setting the isdst field in the tm struct for daylight savings time

#define RV_TM_STANDARD 0 Standard time

#define RV_TM_DST 1 Daylight Savings time

#define RV_TM_UNKNOWN (-1) Unknown (error or let library calculate)

TM FUNCTIONS

RvTmConvertToTime()

DESCRIPTION

Converts calendar time into standard time (seconds and nanoseconds since 1970.)

This function also normalizes the values of the calendar object (which might be the sole purpose for calling it). For example, if the value of seconds is set to 65, it will be normalized to a value of 5 and minutes will be incremented along with any other values that might be affected by it.

SYNTAX

```
RvStatus RvTmConvertToTime(
  IN  RvTm         *tm,
  IN  RvLogMgr*    logMgr,
  OUT RvTime       *t);
```

PARAMETERS

tm

A pointer to the calendar time that is to be converted.

logMgr

The log manager instance, or NULL.

t

A pointer to time structure where the result will be stored.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

The date range of calendar times is larger than the range that can be represented by time. If the calendar time falls outside of this range, the time result will not be correct (but the calender time will still be normalized properly).

RvTmConstructUtc()

DESCRIPTION

Creates a calendar time object representing UTC time based on a standard time.

SYNTAX

```
RvStatus RvTmConstructUtc(
  IN  RvTime*      t,
  IN  RvLogMgr*    logMgr,
  OUT RvTm*        tm);
```

PARAMETERS

t

A pointer to the time structure containing the time.

logMgr

The log manager instance, or NULL.

tm

A pointer to the calendar time object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

Some systems do not support time zones, thus UTC time may actually be local time, depending on how the clock was set.

RvTmConstructLocal()

DESCRIPTION

Creates a calendar time object representing local time based on a standard time.

SYNTAX

```
RvStatus RvTmConstructLocal(
  IN  RvTime*      t,
  IN  RvLogMgr*    logMgr,
  OUT RvTm*        tm);
```

PARAMETERS

t

A pointer to the time structure containing the time.

logMgr

The log manager instance, or NULL.

tm

A pointer to the calendar time object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- Some systems do not support time zones and local time will report the same time as UTC time.
- Some systems do not support daylight savings time properly and will not automatically adjust for it.

RvTmAsctime()

DESCRIPTION

Creates a character string representation of calendar time. Equivalent to the ANSI asctime() function.

SYNTAX

```
RvChar *RvTmAsctime(
  IN  RvTm*        tm,
  IN  RvSize_t     bufsize,
  IN  RvLogMgr*    logMgr,
  OUT RvChar*      buf);
```

PARAMETERS

tm

A pointer to a calendar time object from which to create the string.

bufsize

The size of the buffer (must be at least of the size RV_TM_ASCTIME_BUFSIZE).

logMgr

The log manager instance, or NULL.

buf

A pointer to the buffer to which the string will be copied.

RETURN VALUES

Returns a pointer to the string buffer.

RvTmStrftime()

DESCRIPTION

Creates a custom character string representation of calendar time. Equivalent to the ANSI strftime() function.

SYNTAX

```
RvSize_t RvTmStrftime(
  IN  RvTm*        tm,
  IN  RvChar*      format,
  IN  RvSize_t     maxSize,
  IN  RvLogMgr*    logMgr,
  OUT RvChar*      result);
```

PARAMETERS

tm

A pointer to the calendar time object from which to create the string.

format

The standard ANSI strftime() formatting string.

maxsize

The size of the result buffer.

logMgr

The log manager instance, or NULL.

result

A pointer to the buffer where the resulting string should be copied.

RETURN VALUES

Returns the size of the string that was generated (0 if there was a problem).

REMARKS

Only ANSI standard formats in the string format should be used to insure compatibility across different systems.

3

MEMORY HANDLING MODULE

Memory handling of the Common Core allows the allocation and de-allocation of memory for the use of the Stacks.

Memory handling is built as a memory module which the Stacks use directly (this module does not have to be ported), and a set of drivers that this memory module uses. The osmem driver is the only driver that has to be ported.

OSMEM MODULE

You can view memory handling as simple malloc() and free() capabilities.

You can find the module in *ccore/memdrivers/rvosmem.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- Allocated memory is available from all of the threads and can be freed from any of the threads.
- Allocated memory is available until it is freed, even if the thread that allocated the memory has exited.

The functions included in this chapter are listed below.

MEMORY HANDLING FUNCTIONS

- RvOsMemConstruct()
- RvOsMemDestruct()
- RvOsMemAlloc()
- RvOsMemFree()
- RvOsMemGetInfo()

MEMORY HANDLING FUNCTIONS

RvOsMemConstruct()

DESCRIPTION

Constructs a memory region for use by the Stacks.

This function creates a memory allocator instance which is then used to allocate and free memory, while preserving statistics on the allocated memory itself.

SYNTAX

```
RvStatus RvOsMemConstruct(
  IN  void*        start,
  IN  RvSize_t     size,
  IN  RvSize_t     overhead,
  IN  RvMemory*    moremem,
  IN  void*        attr
  OUT void*        driverRegion);
```

PARAMETERS

start

Not used.

size

Not used.

overhead

The overhead of each allocation. Used for statistical information only.

moremem

Not used.

attr

Not used.

driverRegion

The region being constructed. Used mostly to hold statistical information and other operating system-specific information.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvOsMemDestruct()

DESCRIPTION

Destructs a system driver memory region. If statistical information is required, it destructs the statistics area and its lock.

SYNTAX

```
RvStatus RvOsMemDestruct(
  IN void* driverRegion);
```

PARAMETERS

driverRegion

Pointer to the region object to be destructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvOsMemAlloc()

DESCRIPTION

Allocates memory from a system driver region. Uses the underlying operating system services to allocate a block of memory from the system heap. The equivalent of malloc() in ANSI C.

SYNTAX

```
RvStatus RvOsMemAlloc(
  IN  void        *driverRegion,
  IN  RvSize_t    size,
  OUT void        **result);
```

PARAMETERS

driverRegion

Pointer to the region object.

size

Memory needed in bytes.

result

A pointer to the resulting allocation (the return value of the equivalent malloc() call).

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvOsMemFree()

DESCRIPTION

Frees memory that was allocated from a system driver region. Uses the underlying operating system services to free a block of memory back to the system heap. The equivalent of free() in ANSI C.

SYNTAX

```
RvStatus RvOsMemFree(
  IN void* driverRegion,
  IN void* ptr);
```

PARAMETERS

driverRegion

Pointer to the region object.

ptr

Pointer to the allocated memory to be released.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvOsMemGetInfo()

DESCRIPTION

Returns the collected statistical information about a memory region.

SYNTAX

```
RvStatus RvOsMemGetInfo(
  IN  void*           driverRegion,
  OUT RvMemoryInfo*   meminfo);
```

PARAMETERS

driverRegion

Pointer to the region object where statistical data is stored.

meminfo

Statistical information returned by this function.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

This function does not have to be ported, although it is advised to port it for debugging and maintenance purposes.

4

THREADING MODULES

The Threading modules are responsible for providing facilities that allow thread-safety and multi-threading support by the various Stacks. If the Stack being used can operate as a single-thread Stack and the application itself accesses the Stack from a single thread, this set of modules does not have to be ported.

The macro RV_THREADNESS_TYPE defines whether threading module functionalities should be included into the binary image. The RV_THREADNESS_TYPE value can be defined as one of the following:

- RV_THREADNESS_MULTI

  If the RV_THREADNESS_TYPE value is set to RV_THREADNESS_MULTI, the Stack can operate as a single-thread or multi-threaded Stack and the application will be able to access the Stack from any thread.

- RV_THREADNESS_SINGLE

  If the RV_THREADNESS_TYPE value is set to RV_THREADNESS_SINGLE, the Stack will operate as a single-thread Stack and the application should access the Stack only from the thread that initialized the Stack.

The following functions are included in this chapter:

SEMAPHORE MODULE

Semaphore Constants and Type Definitions

- RvSemaphore
- RvSemaphoreAttr

Semaphore Functions

- RvSemaphoreConstruct()
- RvSemaphoreDestruct()
- RvSemaphorePost()
- RvSemaphoreWait()
- RvSemaphoreSetAttr()
- RvSemaphoreTryWait()

LOCK MODULE

Lock Constants and Type Definitions

- RvLock
- RvLockAttr

Lock Functions

- RvLockConstruct()
- RvLockDestruct()
- RvLockGet()
- RvLockRelease()
- RvLockSetAttr()

MUTEX MODULE

Mutex Constants and Type Definitions

- RvMutex
- RvMutexAttr

Mutex Functions

- RvMutexConstruct()
- RvMutexDestruct()
- RvMutexLock()
- RvMutexUnlock()
- RvMutexSetAttr()

THREAD MODULE

Thread Constants and Type Definitions

- RvThread
- RvThreadId
- RvThreadAttr
- RvThreadFunc

Thread Functions

- RvThreadConstruct()
- RvThreadConstructFromUserThread()
- RvThreadDestruct()
- RvThreadCreate()
- RvThreadStart()
- RvThreadCurrent()
- RvThreadCurrentId()
- RvThreadIdEqual()
- RvThreadSleep()
- RvThreadNanosleep()
- RvThreadGetOsName()
- RvThreadSetPriority()
- RvThreadSetStack()

SEMAPHORE MODULE

The Semaphore module provides semaphore functions and can be found in *ccore/rvsemaphore.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- A semaphore object cannot be destroyed when it has a thread suspended on it.
- Most of the RADVISION Stacks tend to use many semaphores. This could make your porting process a bit more complex if the operating system you are porting to has limited semaphore resources.

Semaphore Constants and Type Definitions

RvSemaphore

A counting semaphore object. See the definitions in *rvsemaphore.h*. The syntax of this object varies between operating systems.

RvSemaphoreAttr

Operating system-specific attributes and options used for semaphore. The syntax of this object varies between operating systems.

SEMAPHORE FUNCTIONS

RvSemaphoreConstruct()

DESCRIPTION

Creates a counting semaphore object.

SYNTAX

```
RvStatus RvSemaphoreConstruct(
  IN  RvUint32       startcount,
  IN  RvLogMgr       *logMgr,
  OUT RvSemaphore    *sema);
```

PARAMETERS

startcount

The initial value of the counter of the semaphore.

logMgr

The log manager instance, or NULL.

sema

A pointer to the semaphore object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

The maximum value of a sempahore is operating system- and architecture-dependent.

RvSemaphoreDestruct()

DESCRIPTION

Destroys a counting semaphore object.

SYNTAX

```
RvStatus RvSemaphoreDestruct(
  IN RvSemaphore    *sema,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

sema

A pointer to the semaphore object to be destructed.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

Never destroy a semaphore object which has a thread suspended on it.

RvSemaphorePost()

DESCRIPTION

Increments the semaphore counter.

SYNTAX

```
RvStatus RvSemaphorePost(
  IN RvSemaphore    *sema,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

sema

A pointer to the semaphore object the counter of which will be incremented.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

The maximum value of a sempahore is operating system- and architecture-dependent.

RvSemaphoreWait()

DESCRIPTION

Decrements a semaphore counter. If the value of the semaphore is 0, the semaphore will suspend the calling task until the semaphore counter becomes greater than null and can be decremented.

SYNTAX

```
RvStatus RvSemaphoreWait(
  IN RvSemaphore    *sema,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

sema

A pointer to the semaphore object to be decremented.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvSemaphoreSetAttr()

DESCRIPTION

Sets the options and attributes to be used when creating and using semaphore objects.

SYNTAX

```
RvStatus RvSemaphoreSetAttr(
  IN RvSemaphoreAttr    *attr,
  IN RvLogMgr           *logMgr);
```

PARAMETERS

attr

A pointer to operating system-specific semaphore attributes to begin using.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- This is a non-reentrant function. Do not call when other threads may be calling rvsemaphore functions.
- These attributes are global and will affect all semaphore functions called thereafter.
- The default values for these attributes are set in *rvccoreconfig.h*.

RvSemaphoreTryWait()

DESCRIPTION

Decrements a semaphore counter.

If the value of the semaphore is 0, the function will not suspend the calling task. Instead, it will return an error code as follows:

- Library part of the error code: RV_ERROR_LIBCODE_CCORE
- Module part of the error code: RV_CCORE_MODULE_SEMAPHORE
- Specific error: RV_ERROR_TRY_AGAIN

SYNTAX

```
RvStatus RvSemaphoreTryWait(
  IN RvSemaphore    *sema,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

sema

A pointer to the semaphore object to be decremented.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

LOCK MODULE

The Lock module provides non-recursive locking functions to use specifically for locking code sections.

You can find this module in *ccore/rvlock.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- These locks are non-recursive. If they are locked twice from the same thread, their behavior is unknown (they may work, crash or deadlock).
- A lock object cannot be destroyed when it has a thread suspended on it.

Lock Constants and Type Definitions

RvLock

A non-recursive lock object. See the definitions in *rvlock.h*. The syntax of this object varies between operating systems.

RvLockAttr

Operating system-specific attributes and options used for locks. The syntax of this object varies between operating systems.

LOCK FUNCTIONS

RvLockConstruct()

DESCRIPTION

Creates a locking object.

SYNTAX

```
RvStatus RvLockConstruct(
  IN  RvLogMgr    *logMgr,
  OUT RvLock      *lock);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

lock

A pointer to the lock object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvLockDestruct()

DESCRIPTION

Destroys a locking object.

SYNTAX

```
RvStatus RvLockDestruct(
  IN RvLock      *lock,
  IN RvLogMgr    *logMgr);
```

PARAMETERS

lock

A pointer to the lock object to be destructed.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

Never destroy a lock object which has a thread suspended on it.

RvLockGet()

DESCRIPTION

Acquires a lock. Will suspend the calling task until the lock is available.

SYNTAX

```
RvStatus RvLockGet(
  IN RvLock      *lock,
  IN RvLogMgr    *logMgr);
```

PARAMETERS

lock

A pointer to the lock object to be acquired.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvLockRelease()

DESCRIPTION

Releases a lock.

SYNTAX

```
RvStatus RvLockRelease(
  IN RvLock      *lock,
  IN RvLogMgr    *logMgr);
```

PARAMETERS

lock

A pointer to the lock object to be released.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

Do not release a lock more times than it has been acquired.

RvLockSetAttr()

DESCRIPTION

Sets the options and attributes to be used when creating and using lock objects.

SYNTAX

```
RvStatus RvLockSetAttr(
  IN RvLockAttr    *attr,
  IN RvLogMgr      *logMgr);
```

PARAMETERS

attr

A pointer to operating system-specific lock attributes to begin using.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Always returns RV_OK.

REMARKS

- This is a non-reentrant function. Do not call when other threads may be calling rvlock functions.
- These attributes are global and will affect all lock functions called thereafter.
- The default values for these attributes are set in *rvccoreconfig.h*.

MUTEX MODULE

The Mutex module provides recursive locking functions to use specifically for locking code sections.

If your operating system does not support this type of locking functionality, you can use the RV_MUTEX_MANUAL interface already implemented in the code (and needs the semaphore module as part of its implementation).

You can find this module in *ccore/rvmutex.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- These mutexes are recursive. It should be possible to lock a mutex more than once from the same thread without deadlocking.
- A mutex object cannot be destroyed when it has a thread suspended on it.

MUTEX CONSTANTS AND TYPE DEFINITIONS

RvMutex

A recursive mutex object. See the definitions in *rvmutex.h*. The syntax of this object varies between operating systems.

RvMutexAttr

Operating system-specific attributes and options used for mutexes. The syntax of this object varies between operating systems.

MUTEX FUNCTIONS

RvMutexConstruct()

DESCRIPTION

Creates a recursive mutex object.

SYNTAX

```
RvStatus RvMutexConstruct(
  IN  RvLogMgr*    logMgr,
  OUT RvMutex*     mu);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

mu

A pointer to the mutex object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvMutexDestruct()

DESCRIPTION

Destroys a recursive mutex object.

SYNTAX

```
RvStatus RvMutexDestruct(
  IN RvMutex*     mu,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

mu

A pointer to the recursive mutex object to be destructed.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

Never destroy a mutex object which has a thread suspended on it.

RvMutexLock()

DESCRIPTION

Acquires a recursive mutex. Will suspend the calling task until the mutex is available.

SYNTAX

```
RvStatus RvMutexLock(
  IN RvMutex*     mu,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

mu

A pointer to the mutex object to be acquired.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvMutexUnlock()

DESCRIPTION

Unlocks a recursive mutex.

SYNTAX

```
RvStatus RvMutexUnlock(
  IN RvMutex*     mu,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

mu

A pointer to the mutex object to be unlocked.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvMutexSetAttr()

DESCRIPTION

Sets the options and attributes to be used when creating and using mutex objects.

SYNTAX

```
RvStatus RvMutexSetAttr(
  IN RvMutexAttr    *attr,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

attr

A pointer to the operating system-specific mutex attributes to begin using.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- This is a non-reentrant function. Do not call when other threads may be calling rvmutex functions.
- These attributes are global and will affect all mutex functions called thereafter.
- The default values for these attributes are set in *rvccoreconfig.h*.

THREAD MODULE

The Thread module provides functions for creating and manipulating threads.

Threads need to be constructed, created, and then started. Between construction and creation, parameters such as stack size and special attributes can be set. The priority is the only parameter that can be changed after creating a thread.

Pre-existing threads can use the RvThreadConstructFromUserThread() function to create a thread handle so that thread-specific variables and other functions which require a thread handle can be used. Since these threads are already running, attributes of these threads can not be changed and some operating systems do not even allow the priority to be changed. Destructing a thread created in this manner does not delete the thread itself, it just returns it to the state it was in when RvThreadConstructFromUserThread() was called. In addition, RvThreadDestruct() **must** be called before such a thread is allowed to exit.

Functions that do not require a thread handle (except RvThreadCurrent()) can be called from any thread, even if a handle was not constructed.

This module has functions that are not described here since these functions do not have to be ported.

You can find this module in *ccore/rvthread.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- Some of the functions in the list below do not have to be ported if the Stack is only single-threaded or thread-safe—they are necessary only for multi-threaded Stacks.

Thread Constants and Type Definitions

RvThread

A thread object.

Syntax

```
typedef struct
{
  RvInt32 state;
  RvChar name[RV_THREAD_MAX_NAMESIZE];
  void *stackaddr;
  void *stackstart;
  RvInt32 reqstacksize;
  RvInt32 realstacksize;
  RvInt32 stacksize;
  RvBool stackallocated;
  RvBool waitdestruct;
  RvLock datalock;
  RvSemaphore exitsignal;
  RvThreadFunc func;
  void *data;
  RvInt32 priority;
  RvThreadAttr attr;
  RvThreadId id;
  RvThreadBlock tcb;
  RvBool autodelete;
  RvThreadFunc exitfunc;
  void *exitdata;
  void* logMgr;
  RvLogSource* threadSource;
  void *vars[RV_THREAD_MAX_VARS];
};
```

Parameters

state

The current thread state.

name

The full name of the thread.

stackaddr

The pointer to the memory for the thread's Stack space.

stackstart

The actual pointer to the start of the Stack (told to the operating system).

reqstacksize

The requested size of the Stack.

realstacksize

The size of the stack told to the operating system (starting at *stackstart*).

stacksize

The actual size of the Stack memory (pointed to by *stackaddr*).

stackallocated

Set to RV_TRUE if the Stack was allocated internally.

waitdestruct

Set to RvTrue if the destruct is waiting for the thread to exit the datalock lock for access to the thread structure.

exitsignal

Used to signal thread completion.

func

The function to call in the thread.

data

The data parameter to be passed to *func*.

priority

The priority of the thread.

attr

The operating system-specific thread attributes.

id

The operating system ID of the thread.

tcb

The operating system-specific task control block or pointer to it.

autodelete

Specifies whether or not this thread is automatically deleted if discovered that it was stopped. **Default:** RV_FALSE

exitfunc

The function to call on thread exit.

exitdata

The parameter to pass to exitfunc call.

logMgr

The log manager instance.

threadSource

The thread module log source.

vars

The values of the thread-specific variables.

RvThreadId

Operating system-specific thread ID. Used to identify threads regardless of whether or not a thread handle has been constructed for it. See the definitions in *rvthread.h.* The syntax of this object varies between operating systems.

RvThreadAttr

Operating system-specific attributes and options used for threads. The syntax of this object varies between operating systems.

RvThreadFunc

This function should be called as the main function of a thread.

Syntax

```
void RvThreadFunc(RvThread *th, void *data);
```

Parameters

th

A pointer to the thread structure of the current thread.

data

The user parameter that was passed to RvThreadConstruct().

THREAD FUNCTIONS

RvThreadConstruct()

DESCRIPTION

Constructs a thread object.

An actual thread is not created yet. All settings for this thread use the default parameters defined in *rvccoreconfig.h*. Therefore, it is not necessary to set every parameter if the defaults are acceptable for this thread.

SYNTAX

```
RvStatus RvThreadConstruct(
  IN  RvThreadFunc    func,
  IN  void            *data,
  IN  RvLogMgr        *logMgr,
  OUT RvThread        *th);
```

PARAMETERS

func

A pointer to the function that will be executed in the new thread.

data

User-defined data that will be passed to the function when it is executed.

logMgr

The log manager instance, or NULL.

th

A pointer to the thread object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvThreadConstructFromUserThread()

DESCRIPTION

Constructs a thread object and associates it with an existing thread that was created externally (not using RvThreadConstruct()).

SYNTAX

```
RvStatus RvThreadConstructFromUserThread(
  IN  RvLogMgr    *logMgr,
  OUT RvThread    *th);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

th

A pointer to the thread object to be constructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

This function should be called from the thread itself. Threads which have made this call must call RvThreadDestruct() before exiting.

RvThreadDestruct()

DESCRIPTION

Destructs a thread object.

Threads cannot be destructed until they have exited. Therefore, if the thread has not exited, this function will wait until the thread exits before returning.

Threads created with RvThreadConstructFromUserThread() are a special case in which RvThreadDestruct() **must** be called from the thread itself before exiting (and obviously will not wait for the thread to exit).

SYNTAX

```
RvStatus RvThreadDestruct(
  IN RvThread *th);
```

PARAMETERS

th

A pointer to the thread object to be destructed.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

RvThreadDestruct() can only be called once on each thread. Therefore, this function may not be called simultaneously from multiple threads (with the same thread to destruct).

RvThreadCreate()

DESCRIPTION

Creates the actual thread.

The thread will be created by the operating system and will allocate all the needed resources but will not begin executing.

SYNTAX

```
RvStatus RvThreadCreate(
  IN RvThread *th);
```

PARAMETERS

th

A pointer to the thread object to be created.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- Some operating systems do not allow threads to be started in the suspended state so the physical thread will not appear until RvThreadStart() is called.
- If the call fails, RvThreadDestruct() should be called to clean up properly.

RvThreadStart()

DESCRIPTION

Starts the thread execution.

SYNTAX

```
RvStatus RvThreadStart(
  IN RvThread *th);
```

PARAMETERS

th

A pointer to the thread object to be started.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

If the call fails, RvThreadDestruct() should be called to clean up properly.

RvThreadCurrent()

DESCRIPTION

Gets the thread handle of the current thread.

SYNTAX

```
RvThread * RvThreadCurrent(void);
```

PARAMETERS

None.

RETURN VALUES

Returns a pointer to the thread object of the current thread.

REMARKS

If the thread was not created with this Thread module or was not attached to a thread object with RvThreadConstructFromUserThread(), this function will return NULL.

RvThreadCurrentId()

DESCRIPTION

Gets the operating system-specific thread ID of the current thread.

SYNTAX

```
RvThreadId RvThreadCurrentId(void);
```

PARAMETERS

None.

RETURN VALUES

Returns the thread ID of the current thread.

REMARKS

This function can be called for all threads regardless of how they were created.

RvThreadIdEqual()

DESCRIPTION

Compares two thread IDs.

SYNTAX

```
RvBool RvThreadIdEqual(
  IN RvThreadId id1,
  IN RvThreadId id2);
```

PARAMETERS

id1

A thread ID.

id2

A thread ID.

RETURN VALUES

Returns RV_TRUE if the threads are the same, otherwise RV_FALSE.

RvThreadSleep()

DESCRIPTION

Suspends the current thread for the requested amount of time.

SYNTAX

```
RvStatus RvThreadSleep(
  IN const RvTime    *t,
  IN RvLogMgr        *logMgr);
```

PARAMETERS

t

A pointer to the RvTime structure containing the amount of time to sleep.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- This function can be called for all threads regardless of how they were created.
- The exact time of suspension is based on the operating system and the resolution of the system clock.

RvThreadNanosleep()

DESCRIPTION

Suspends the current thread for the requested amount of time.

SYNTAX

```
RvStatus RvThreadNanosleep(
  IN RvInt64      nsecs,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

nsecs

The time to sleep in nanoseconds.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- This function can be called for all threads regardless of how they were created.
- The exact time of suspension is based on the operating system and the resolution of the system clock.

RvThreadGetOsName()

DESCRIPTION

Gets the name of a thread as seen by the operating system.

SYNTAX

```
RvStatus RvThreadGetOsName(
  IN  RvThreadId    id,
  IN  RvInt32       size,
  OUT RvChar        *buf);
```

PARAMETERS

id

The thread ID.

size

The size of the buffer.

buf

A pointer to the buffer where the name will be copied.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

REMARKS

- This function can be called for all threads regardless of how they were created.
- If the buffer is too small for the name, the name will be truncated.
- Only works on threads that exist (are executing).

RvThreadSetPriority()

DESCRIPTION

Sets the Stack priority for a thread.

SYNTAX

```
RvStatus RvThreadSetPriority(
  IN RvThread    *th,
  IN RvInt32     priority);
```

PARAMETERS

th

A pointer to the thread object of which the priority will be set.

priority

The priority to which to set the thread.

RETURN VALUES

Returns RV_OK on success, or an error code on failure.

RvThreadSetStack()

DESCRIPTION

Sets the Stack information for a thread.

SYNTAX

```
RvStatus RvThreadSetStack(
  IN RvThread    *th,
  IN void        *stackaddr,
  IN RvInt32     stacksize);
```

PARAMETERS

th

Pointer to the thread object for which the Stack information will be set.

stackaddr

Address of the memory to use for the Stack. If set to NULL, it is allocated.

stacksize

Size of the Stack.

RETURN VALUES

RV_OK if successful, otherwise an error code.

REMARKS

The Stack information can not be changed once a thread has been created. If stackaddr is set to NULL, the amount of Stack space used for internal overhead will be allocated in addition to the requested Stack size. If stackaddr is set to NULL and stacksize is set to 0, the default Stack settings will be used.

5

NETWORKING MODULES

The Networking modules are responsible for all communications with the network interfaces of the operating system. This includes the operating system's TCP and UDP Stacks, DNS lookups, and so on.

The Windows implementation that is supplied uses WinSock 2, while the UNIX implementation relies on BSD. The macro RV_NET_TYPE defines whether networking module functionalities should be included into the binary image.

The RV_NET_TYPE value can be defined as one of the following:

- RV_NET_NONE—The common core will exclude networking functionality from the binary image.
- Bit mask that includes at least one of following:
  - RV_NET_IPV4-IP version 4 is supported.
  - RV_NET_IPV6-IP version 4 and IP version 6 are supported.

The following functions are included in this chapter:

SOCKET MODULE

Socket Constants and Type definitions

- RvSocket
- RvSocketProtocol

Socket Functions

- RvSocketConstruct()

- RvSocketDestruct()
- RvSocketBind()

Socket Parameter Functions

- RvSocketSetBuffers()
- RvSocketSetLinger()
- RvSocketReuseAddr()
- RvSocketSetBlocking()
- RvSocketGetBytesAvailable()
- RvSocketGetLastError()
- RvSocketGetLocalAddress()
- RvSocketGetRemoteAddress()
- RvSocketSetTypeOfService()
- RvSocketGetTypeOfService()
- RvSocketSetBroadcast()
- RvSocketSetMulticastTtl()
- RvSocketSetMulticastInterface()
- RvSocketJoinMulticastGroup()
- RvSocketLeaveMulticastGroup()

Connection Oriented Socket Functions

- RvSocketConnect()
- RvSocketAccept()
- RvSocketListen()
- RvSocketShutdown()
- RvSocketSendBuffer()
- RvSocketReceiveBuffer()

SELECT MODULE

Select Constants and Type Definitions

- RvSelectEvents
- RvSelectEngine
- RvSelectFd

- RvSelectConnect
- RvSelectAccept
- RvSelectClose
- RvSelectRead
- RvSelectWrite

Select Callbacks

- RvSelectCb()
- RvSelectPreemptionCb()
- RvTimerFunc()

Select Functions

- RvSelectSetMaxFileDescriptors()
- RvSelectGetMaxFileDescriptors()

Select Engine Functions

- RvSelectConstruct()
- RvSelectDestruct()
- RvSelectWaitAndBlock()

Select Preemption Functions

- RvSelectStopWaiting

Select File Descriptor Functions

- RvFdConstruct()
- RvFdDestruct()
- RvFdGetSocket()
- RvSelectGetEvents()
- RvSelectFindFd()
- RvSelectAdd()
- RvSelectRemove()
- RvSelectUpdate()

Select Timer Management Functions

- RvSelectSetTimeoutInfo()
- RvSelectGetTimeoutInfo()

HOST MODULE

Host Functions

- RvHostLocalGetName()
- RvHostLocalGetAddress()

TLS MODULE

TLS Constants and Type Definitions

- RvCompareCertificateCB()
- RvTLSMethod
- RV_TLS_DEFAULT_CERT_DEPTH
- RvTLSEvents
- RvPrivKeyType
- RvTLSEngine
- RvTLSSession

TLS Functions

- RvTLSEngineConstruct()
- RvTLSEngineAddAutorityCertificate()
- RvTLSEngineAddCertificate()
- RvTLSEngineDestruct()
- RvTLSSessionConstruct()
- RvTLSSessionDestruct()
- RvTLSSessionClientHandshake()
- RvTLSSessionServerHandshake()
- RvTLSSessionReceiveBuffer()
- RvTLSSessionSendBuffer()
- RvTLSSessionShutdown()
- RvTLSTranslateSelectEvents()

- RvTLSTranslateTLSEvents()
- RvTLSGetCertificateLength()
- RvTLSGetCertificate()
- RvTLSSessionGetCertificateLength()
- RvTLSSessionGetCertificate()
- RvTLSGetCertificateVerificationError()
- RvTLSSessionCheckCertAgainstName()
- RvTLSEngineCheckPrivateKey()

SOCKET MODULE

The Socket module provides the socket interface for networking. It is actually a set of functions written on top of socket functions and implemented according to the different operating systems. The Socket module provides both blocking and non-blocking implementations of all socket functions. Before beginning to port this module, verify which socket implementation your Stack needs (blocking, non-blocking or both) and which transport protocols you need to port (IPv6, TCP, UDP, and others). You may not be able to implement all of the functions in this module in all operating systems. Check that the function you are porting can be implemented in the operating system you need to use (for example, some of the socket options are not implemented by all operating systems). If the operating system cannot implement a function, make sure that the function does not have an essential role in the Stack you are using. If it does, supply an empty implementation of the function.

Before porting this module, it is recommended that you also read the Select Module section, since these two modules are closely related and highly interactive.

You can find this module in *ccore/netdrivers/rvsocket.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- The Socket module provides both blocking and non-blocking implementations of all socket functions. When the operating system does not give an implementation of non-blocking sockets, it is implemented by the socket module itself.
- This module uses the Address module, which is not an operating systems-dependent module and is not documented here. The address module contains an RvAddress type definition and several utility functions to access that type. Please refer to *cutils/rvaddress.c* directly to see its implementation if needed.
- Sockets can be accessed, created and destroyed from any thread. If you are using a multi-threaded Stack or writing a multi-threaded application that calls the Stack's API functions from different threads, extra work may be needed to deal with this requirement if your operating system does not support it.

USING SOCKET FUNCTIONS

The following section describes the Socket function groups and how they can be used.

SOCKET FUNCTIONS

To understand how the main functions of the Socket module interact, consider the following life-cycle of an IP socket:

Table 5-1 *IP Socket Life-cycle*

Life Stage	Functions	Description
Initialize Socket Module	RvSocketInit()[1]	This function is called before any other Socket functions are called in order to initialize the Socket module.
Create Socket	RvSocketConstruct() RvSocketBind()	The RvSocketConstruct() function opens an IP socket and RvSocketBind() binds it to an IP address and port number.
Establish Connection	RvSocketConnect() RvSocketListen() RvSocketAccept()	RvSocketConnect() starts a connection between the socket and a specific destination. Alternatively, the socket may listen for incoming connections using RvSocketListen(). Whichever side is listening can then use RvSocketAccept() to establish the socket's connection.

Table 5-1 *IP Socket Life-cycle (Continued)*

Life Stage	Functions	Description
Use Socket	RvSocketSendBuffer()	RvSocketSendBuffer() sends data through the socket.
	RvSocketReceiveBuffer()	RvSocketReceiveBuffer() receives data through the socket.
	RvSocketSetBuffers()	RvSocketSetBuffers() sets the send and receive buffer sizes. In addition, at this stage, the Select Event-Driven functions, Socket Information functions and Socket Multicast functions can be used.
Close Socket	RvSocketShutdown() RvSelectDestruct()	The RvSelectDestruct() function closes the socket. The RvSocketShutdown() function may be used to block the sending of further packets before RvSelectDestruct() actually closes the socket.
Terminate Socket	RvSocketEnd()[2]	The RvSocketEnd() function is called just before the entire application shuts down. It gives you the opportunity to close files or otherwise clean up if necessary.

1. This function is not described here and should be used in the porting process if your operating system requires it.
2. Same as above footnote.

Note For more information, refer to functions in the Select Event-Driven Functions and Interaction Between Socket and Select Event-Driven Functions sections.

Blocking Sockets

Sockets can be defined to work in blocking or non-blocking modes. In blocking mode, the socket will block until a failure or until the operation being blocked on has succeeded.

In non-blocking mode, the function of the socket will return immediately—before the actual completion of the operation initiated by the function of the socket.

To set the mode of operation used by a socket, the function RvSocketSetBlocking() is supplied.

The Select module is used to handle events on non-blocking sockets.

Socket Multicast

Socket Multicast functions are used to prepare the socket for Multicast transmissions:

- RvSocketSetBroadcast()—Sets the permission for sending Multicast data on a socket.
- RvSocketSetMulticastInterface()—Used to set the Multicast interface for the socket.
- RvSocketSetMulticastTtl()—Used to set the Multicast “time-to-live” for the socket.
- RvSocketJoinMulticastGroup()—Used to add IP addresses to the Multicast list.
- RvSocketLeaveMulticastGroup()—Used to remove IP addresses from the Multicast list.

SOCKET CONSTANTS AND TYPE DEFINITIONS

RvSocket

The socket handle of the operating systems. Most operating systems will supply this parameter as an “int”. Windows supplies this parameter as a “SOCKET”. A pointer of this type is supplied to all of the functions of the socket module when dealing with sockets.

RvSocketProtocol

The protocol that the socket uses. Check which of these protocols you need to implement when constructing and binding the socket.

Syntax

```
typedef enum
{
  RvSocketProtocolUdp = 0,
  RvSocketProtocolTcp,
  RvSocketProtocolIcmp,
  RvSocketProtocolSctp
}RvSocketProtocol;
```

Socket Functions

RvSocketConstruct()

Description

Constructs a socket object.

During construction time, the type of address and protocol of the socket must be supplied. A call to RvSocketSetBlocking() must follow the call to this function to set this socket as a blocking or non-blocking socket.

Syntax

```
RvStatus RvSocketConstruct(
  IN  RvInt              addressType,
  IN  RvSocketProtocol   protocolType,
  IN  RvLogMgr*          logMgr,
  OUT RvSocket*          sock);
```

Parameters

addressType

The address type of the created socket.

protocolType

The type of protocol to use (TCP, UDP, etc.). See RvSocketProtocol.

logMgr

The log manager instance, or NULL.

sock

The socket to construct.

Return Values

Returns RV_OK on success, other on failure.

RvSocketDestruct()

DESCRIPTION

Closes a socket.

SYNTAX

```
RvStatus RvSocketDestruct(
  IN RvSocket*       sock,
  IN RvBool          cleanSocket,
  IN RvPortRange*    portRange
  IN RvLogMgr*       logMgr);
```

PARAMETERS

sock

The socket to shutdown.

cleanSocket

RV_TRUE if you want to clean the socket before shutting it down. This will attempt to receive some buffers from the socket. It is suggested to set this value to RV_TRUE for TCP sockets. It should be set to RV_FALSE for UDP sockets.

portRange

The port range to which to return the port of this socket. If NULL, the port of the socket will not be added to any port range object.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

Before porting this function, check if your application needs the *portRange* feature and that the Stack you are using supports this feature.

RvSocketBind()

DESCRIPTION

Binds a socket to a local address.

SYNTAX

```
RvStatus RvSocketBind(
  IN RvSocket*       sock,
  IN RvAddress*      address,
  IN RvPortRange*    portRange
  IN RvLogMgr*       logMgr);
```

PARAMETERS

socket

The socket to bind.

address

The address to which to bind the socket.

portRange

The port range to use if address states an arbitrary port. NULL if this parameter should be ignored.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

Before porting this function, check if your application needs the *portRange* feature and that the Stack you are using supports this feature.

SOCKET PARAMETER FUNCTIONS

The following functions modify or query the parameters of the sockets.

Some of these functions are not portable to all of the operating systems. In such cases, you should probably leave the implementation of the function empty and rely on the default value of the operating system for that parameter.

RvSocketSetBuffers()

DESCRIPTION

Sets the send and receive buffer sizes for the specified socket.

The default sizes of send and receive buffers are platform-dependent. This function can be used to increase the default sizes if larger packets are expected, such as video packets, or to decrease the default size if smaller packets are expected.

SYNTAX

```
RvStatus RvSocketSetBuffers(
  IN RvSocket*    sock,
  IN RvInt32      sendSize,
  IN RvInt32      recvSize
  IN RvLogMgr*    logMgr);
```

PARAMETERS

socket

The socket whose buffer sizes are to be changed.

sendSize

The new size of the send buffer. If the size is negative, the existing value remains.

recvSize

The new size of the receive buffer. If the size is negative, the existing value remains.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSocketSetLinger()

DESCRIPTION

Sets the linger time after the socket is closed.

SYNTAX

```
RvStatus RvSocketSetLinger(
  IN RvSocket*    sock,
  IN RvInt32      lingerTime
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The socket to modify.

lingerTime

The time to linger in seconds. Setting this parameter to –1 sets the linger off for this socket.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function can be called only on TCP sockets.

RvSocketReuseAddr()

DESCRIPTION

Sets the socket as a reusable one (in terms of its address).

This allows a TCP server and UDP multicast addresses to be used by other processes on the same machine as well. This function must be called before RvSocketBind().

SYNTAX

```
RvStatus RvSocketReuseAddr(
  IN RvSocket     *socket,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The socket to modify.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSocketSetBlocking()

DESCRIPTION

Sets the blocking or non-blocking mode on a socket.

This function must be called after calling RvSocketConstruct(), otherwise, the socket's mode of operation will be determined by each operating system separately, causing applications and Stacks to be non-portable.

SYNTAX

```
RvStatus RvSocketSetBlocking(
  IN RvSocket*    sock,
  IN RvBool       isBlocking,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

socket

The socket to modify.

isBlocking

RV_TRUE for a blocking socket. RV_FALSE for a non-blocking socket.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSocketGetBytesAvailable()

DESCRIPTION

Gets the number of bytes that are in the receive buffer of the socket.

This number might be larger than the return result of a single recv() operation. This is important when dealing with data-gram types of connections (such as UDP). This function is not available by some of the operating systems.

SYNTAX

```
RvStatus RvSocketGetBytesAvailable(
  IN  RvSocket*    sock,
  IN  RvLogMgr*    logMgr,
  OUT RvSize_t*    bytesAvailable);
```

PARAMETERS

socket

The socket to check.

logMgr

The log manager instance, or NULL.

bytesAvailable

The number of bytes in the receive buffer of the socket.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function is not available by some of the operating systems.

RvSocketGetLastError()

DESCRIPTION

Gets the last error that occurred on a socket.

This function works for synchronous sockets only. The return value is operating system-specific. A value of 0 means no error occurred.

SYNTAX

```
RvStatus RvSocketGetLastError(
  IN  RvSocket*    sock,
  IN  RvLogMgr*    logMgr,
  OUT RvInt32*     lastError);
```

PARAMETERS

sock

The socket to check.

logMgr

The log manager instance, or NULL.

lastError

The error that occurred. 0 means no error.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSocketGetLocalAddress()

DESCRIPTION

Gets the local address that this socket uses.

SYNTAX

```
RvStatus RvSocketGetLocalAddress(
  IN  RvSocket     *sock,
  IN  RvLogMgr     *logMgr,
  OUT RvAddress    *address);
```

PARAMETERS

sock

The socket to check.

logMgr

The log manager instance, or NULL.

address

The local address. Must be destructed by the caller to this function.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSocketGetRemoteAddress()

DESCRIPTION

Gets the remote address that this socket uses.

SYNTAX

```
RvStatus RvSocketGetRemoteAddress(
  IN  RvSocket     *sock,
  IN  RvLogMgr     *logMgr,
  OUT RvAddress    *address);
```

PARAMETERS

socket

The socket to check.

logMgr

The log manager instance, or NULL.

address

The remote address. Must be destructed by the caller to this function.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function can be called only on UDP sockets.

RvSocketSetTypeOfService()

DESCRIPTION

Sets the type of service (DiffServ Code Point) of the socket (IP_TOS).

SYNTAX

```
RvStatus RvSocketSetTypeOfService(
  IN RvSocket*    sock,
  IN RvInt        typeOfService,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The socket to modify.

typeOfService

The type of service to set.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- This function is supported by few operating systems.
- IPv6 does not support type of service.

RvSocketGetTypeOfService()

DESCRIPTION

Gets the type of service (DiffServ Code Point) of the socket (IP_TOS).

SYNTAX

```
RvStatus RvSocketGetTypeOfService(
  IN  RvSocket*    sock,
  IN  RvLogMgr*    logMgr,
  OUT RvInt32*     typeOfService);
```

PARAMETERS

sock

The socket to modify.

logMgr

The log manager instance, or NULL.

typeOfService

The type of service to set.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- This function is supported by few operating systems.
- IPv6 does not support type of service.

RvSocketSetBroadcast()

DESCRIPTION

Sets permission for sending broadcast datagrams on a socket.

SYNTAX

```
RvStatus RvSocketSetBroadcast(
  IN RvSocket*    sock,
  IN RvBool       canBroadcast,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The socket to modify.

canBroadcast

RV_TRUE for permitting the broadcast. RV_FALSE for not permitting the broadcast.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function is needed only if the Stack that is being ported supports multicast sending.

RvSocketSetMulticastTtl()

DESCRIPTION

Sets the TTL to use for multicast sockets (UDP).

SYNTAX

```
RvStatus RvSocketSetMulticastTtl(
  IN RvSocket*    sock,
  IN RvInt32      ttl,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The socket to modify.

ttl

The TTL to set.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function is needed only if the Stack that is being ported supports multicast sending.

RvSocketSetMulticastInterface()

DESCRIPTION

Sets the interface to use for multicast packets (UDP).

SYNTAX

```
RvStatus RvSocketSetMulticastInterface(
  IN RvSocket*     sock,
  IN RvAddress*    address,
  IN RvLogMgr*     logMgr);
```

PARAMETERS

socket

The socket to modify.

address

The local address to use for the interface.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function is needed only if the Stack that is being ported supports multicast sending.

RvSocketJoinMulticastGroup()

DESCRIPTION

Joins a multicast group.

SYNTAX

```
RvStatus RvSocketJoinMulticastGroup(
  IN RvSocket*     sock,
  IN RvAddress*    multicastAddress,
  IN RvAddress*    interfaceAddress,
  IN RvLogMgr*     logMgr);
```

PARAMETERS

sock

The socket to modify.

multicastAddress

The multicast address to join.

interfaceAddress

The interface address to use on the local host. Setting this parameter to NULL chooses an arbitrary interface.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- This function is needed only if the Stack that is being ported supports multicast sending.
- This function can be called only on UDP sockets.

RvSocketLeaveMulticastGroup()

DESCRIPTION

Leaves a multicast group.

SYNTAX

```
RvStatus RvSocketLeaveMulticastGroup(
  IN RvSocket*     sock,
  IN RvAddress*    multicastAddress,
  IN RvAddress*    interfaceAddress,
  IN RvLogMgr*     logMgr);
```

PARAMETERS

sock

The socket to modify.

multicastAddress

The multicast address to leave.

interfaceAddress

The interface address to use on the local host. Setting this paramours to NULL chooses an arbitrary interface.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- This function is needed only if the Stack that is being ported supports multicast sending.
- This function can be called only on UDP sockets.

CONNECTION ORIENTED SOCKET FUNCTIONS

These functions should be used when the protocol of the socket requires a connection to be established prior to sending and receiving any data packets. TCP is such a protocol.

RvSocketConnect()

DESCRIPTION

Starts a connection between the specified socket and the specified destination.

The destination must be running the RvSocketListen() function to receive the incoming connection. In blocking mode, this function returns only when the socket is connected, or on a timeout with a failure return value. In non-blocking mode, this function returns immediately, and when the remote side accepts this connection, this socket will receive the RvSelectConnect event in the Select module.

SYNTAX

```
RvStatus RvSocketConnect(
  IN RvSocket*     sock,
  IN RvAddress*    address,
  IN RvLogMgr*     logMgr);
```

PARAMETERS

sock

The socket to connect.

address

The remote address to which to connect this socket.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function needs to be ported only if the Stack uses TCP.

RvSocketAccept()

DESCRIPTION

Accepts an incoming socket connect request, creating a new socket object.

In blocking mode, this function blocks until an incoming connection request to this socket is made. In non-blocking mode, this function will exit immediately and when an incoming connection request to this socket is made, this socket will receive the RvSelectAccept event in the Select module.

SYNTAX

```
RvStatus RvSocketAccept(
  IN  RvSocket*     sock,
  IN  RvLogMgr*     logMgr,
  OUT RvSocket*     newSocket,
  OUT RvAddress*    remoteAddress);
```

PARAMETERS

sock

The listening socket receiving the incoming connection.

logMgr

The log manager instance, or NULL.

newSocket

Accepted socket information.

remoteAddress

The address of the remote side of connection. Can be passed as NULL if not needed.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSocketListen()

DESCRIPTION

Listens on the specified socket for connections from other locations.

In non-blocking mode, when a connection request is received, this socket will receive the RvSelectAccept event in the Select module.

SYNTAX

```
RvStatus RvSocketListen(
  IN RvSocket     *socket,
  IN RvUint32     queuelen,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The listening socket receiving the incoming connection.

queuelen

The length of the queue of the pending connections.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function needs to be ported only if the Stack uses TCP.

RvSocketShutdown()

DESCRIPTION

Shuts down a TCP socket.

This function should be called before calling RvSocketDestruct() for TCP sockets. In blocking mode, this function blocks until RvSocketDestruct() can be called in a graceful manner to close the connection.

In non-blocking mode, when the remote side closes its connection, this socket will receive the RvSelectClose event in the Select module and RvSocketDestruct() should be called at that point.

SYNTAX

```
RvStatus RvSocketShutdown(
  IN RvSocket*    socket,
  IN RvBool       cleanSocket,
  IN RvLogMgr*    logMgr);
```

PARAMETERS

sock

The socket to shut down.

cleanSocket

RV_TRUE if you want to clean the socket before shutting it down. This will try to receive data from the socket. It is suggested to set this value to RV_TRUE for non-blocking sockets.

logMgr

The log manager instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function needs to be ported only if the Stack uses TCP.

Send and Receive Socket Functions

These functions are used to send data on sockets to remote applications and receive data on sockets from remote applications. They are the reason the Socket module actually exists.

RvSocketSendBuffer()

DESCRIPTION

Sends a buffer on a socket.

This function actually copies the given buffer to the memory of the operating system for sending at a later time. In blocking mode this function will wait until there is enough buffer space within the transport system to hold the data to be transmitted. In non-blocking mode, this function will return immediately, indicating the exact amount of bytes the operating system has processed and sent.

SYNTAX

```
RvStatus RvSocketSendBuffer(
  IN  RvSocket*     sock,
  IN  RvUint8*      buffer,
  IN  RvSize_t      bytesToSend,
  IN  RvAddress*    remoteAddress,
  IN  RvLogMgr*     logMgr,
  OUT RvSize_t*     bytesSent);
```

PARAMETERS

sock

The socket on which to send the buffer.

buffer

The buffer to send.

bytesToSend

The number of bytes to send from the buffer.

remoteAddress

Address to which to send the buffer. Should be set to NULL on connection-oriented sockets (TCP).

logMgr

The log manager instance, or NULL.

bytesSent

The number of bytes that were sent. If less than *bytesToSend*, the application should wait for the RvSelectWrite event in the Select Module before trying to send the buffer again.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function handles both TCP and UDP sockets. You might have to port only parts of this function if you need only one of these protocols.

RvSocketReceiveBuffer()

DESCRIPTION

Receives a buffer from a socket.

In blocking mode, this function will block until a buffer is received on the socket. In non-blocking mode, this function returns immediately, even if there is nothing to be received on this socket at the moment. This function is usually called after the RvSelectRead event is received in the Select Module on this socket, indicating that there is information to receive on this socket.

SYNTAX

```
RvStatus RvSocketReceiveBuffer(
  IN  RvSocket*     sock,
  IN  RvUint8*      buffer,
  IN  RvSize_t      bytesToReceive,
  IN  RvLogMgr*     logMgr,
  OUT RvSize_t*     bytesReceived,
  OUT RvAddress*    remoteAddress);
```

PARAMETERS

sock

The socket from which to receive the buffer.

buffer

The buffer to use for received data.

bytesToReceive

The number of bytes available on the given buffer. For UDP sockets, this value must be higher than the incoming datagram.

logMgr

The log manager instance, or NULL.

bytesReceived

The number of bytes that were actually received.

remoteAddress

The address from which the buffer was received. Should be given as NULL on connection-oriented sockets (TCP). This address is constructed by this function and should be destructed by the caller to this function.

Return Values

Returns RV_OK on success, or a negative value on failure.

Remarks

This function handles both TCP and UDP sockets. You might have to port only parts of this function if you need only one of these protocols.

SELECT MODULE

The Select module is responsible for handling events on sockets working asynchronously. The module is actually a wrapper for the BSD select() with some enhancements. The Select module allows you to work with interfaces other than the BSD select interfaces, such as WSASelect() in Windows, or poll() and /dev/poll on operating systems that support these interfaces.

The Select module initiates a timer heap (RvTimerQueue) defined by cbase/rvtimer.h. The timer heap is used by the Stacks to manage timeouts. In addition, the Stack can set timeout parameters directly when applying the core Select module Wait and Block functionality. The RvTimer module is not described in the Porting Guide because it is operating system-independent.

The core handles only one select engine, which is defined in the specific thread. For this thread, the select engine is allocated and constructed during the first call to the RvSelectConstruct() function in that thread. The constructed select engine is saved into the TLS (Thread Local Storage) of the thread. In case the RvSelectConstruct function is called again from the same thread, it will return the previously allocated and initiated select engine rather than allocate a new one. RvSelectDestruct() must be called from the same thread from which RvSelectConstruct() was called.

This module is more interface-oriented than operating system-oriented. Your operating system of choice might have implementations of several interfaces that can match the requirements of the Select module. You are encouraged to use the best one that suits you from a performance point of view (the number of sockets being processed concurrently, speed, etc.).

Before porting this module, it is advisable that you also read the Socket Module, since they are closely related and highly interact with each other.

You can find this module in *ccore/rvselect.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- The Select module is used for the purpose of sockets only. It is not used for real files.
- Your operating system of choice may not support Close, Accept and Connect events. If this is the case, you need to translate the events of the operating system into these events.
- A select engine is constructed and destructed in a single, specific thread—even in multi-threaded applications. Other functionalities of the engine can be used from threads other than the one in which they were created.

Select Event-Driven Functions

The RvSelectAdd(), RvSelectUpdate() and RvSelectRemove() functions are part of an event-driven mechanism. In an event-driven mechanism, the events expected to occur are registered with RvSelectAdd() and when the registered events occur, an event-handler function is called.

Most events indicate that a particular action is possible and that the registered callback routine can perform the action. For example, RvSelectAccept indicates that the socket connection is ready to be accepted and that the registered callback routine can call the RvSocketAccept() function.

Table 5-2 describes events and callback routine responses:

Table 5-2 *Events and Event Handlers*

Event	Description	Event-Handler Function response
RvSelectAccept	A socket connection is waiting to be accepted.	RvSocketAccept()
RvSelectRead	The socket has data that is ready to be read.	RvSocketReceiveBuffer()
RvSelectWrite	The socket is ready for data to be written.	RvSocketSendBuffer()
RvSelectClose	'The connection identified by the socket has been closed.	RvSocketDestruct()

The RvSelectConnect event follows RvSocketConnect() and indicates whether or not a connection has taken place. The event-handler function that follows RvSelectConnect can perform any action required on the socket.

Interaction Between Socket and Select Event-Driven Functions

To understand how Socket functions and Select event-driven functions interact, consider the following life-cycle of a TCP/IP socket between a Server and a Client:

Table 5-3 *TCP/IP Socket Life-cycle*

Step	Server	Client	Explanation
1	RvSocketConstruct() RvSocketBind()	RvSocketConstruct() RvSocketBind()	Both the Server and the Client open a local IP socket.
2	RvSocketListen()	RvSocketConnect()	The Server listens for incoming connections. The Client initiates a connection to the Server, causing an RvSelectAccept event at the Server.
3	RvSocketAccept()		The Server accepts the connection, causing an RvSelectConnect event at the Client.

Table 5-3 *TCP/IP Socket Life-cycle*

Step	Server	Client	Explanation
4	RvSocketSendBuffer()	RvSocketReceiveBuffer()	When the Server receives a RvSelectWrite event, it writes data to the socket. The Client receives an RvSelectRead event and reads the data from the socket.
5		RvSocketShutdown()	The Client blocks receipt of packets on the socket, initiating a graceful close of the socket. This causes a RvSelectClose event at the Server. Alternatively, the Server could have used RvSocketShutdown(), which would have caused an RvSelectClose event at the Client.
6	RvSocketDestruct()	RvSocketDestruct()	The Server closes the socket, causing an RvSelectClose event at the Client. The Client closes the socket.

Note For more information, refer to functions in the Socket Functions section.

SELECT CONSTANTS AND TYPE DEFINITIONS

RvSelectEvents

The type declaration of events that can be waited for on file descriptors.

Not all operating systems support all of these events. In this case, the user of this module is responsible for resolving the actual event from the events that were received. All operating systems support read and write events.

Note Some operating systems (mainly UNIX) may not return CLOSE events. In this case, receiving a READ event, and actually trying to read, will cause an error which can be interpreted as a CLOSE event.

Syntax

```
typedef struct RvSelectFdInternal RvSelectFd;
```

RvSelectConnect

A request to a non-blocking RvSocketConnect() has succeeded or failed. When receiving this event, the user has to check the error parameter to see whether this socket is connected.

RvSelectAccept

A non-blocking socket that called RvSocketListen() has an incoming connection waiting to be accepted. The user should call RvSocketAccept() when receiving this event.

RvSelectClose

The remote side of this socket has been closed. RvSocketDestruct() should be called on this socket to make sure it releases any used resources.

RvSelectRead

A non-blocking socket received data from a remote machine and RvSocketReceiveBuffer() should be called to retrieve this data. If all the data is not read, this event will automatically be involved again in the future.

RvSelectWrite

A non-blocking socket is ready to send data.

At this stage, the application can call RvSocketSendBuffer() to send any pending outgoing data. Note that RvSocketSendBuffer() might not be able to send the data although RvSelectWrite was indicated on the socket.

Syntax

```
typedef RvUint16 RvSelectEvents;
```

Possible values:

```
#define RvSelectRead        RV_SELECT_READ
#define RvSelectWrite       RV_SELECT_WRITE
#define RvSelectAccept      RV_SELECT_ACCEPT
#define RvSelectConnect     RV_SELECT_CONNECT
#define RvSelectClose       RV_SELECT_CLOSE
```

RvSelectEngine

The Fd Events engine declaration. This structure holds the parameters needed to operate a select() function call (or other such interfaces). The contents are interface-dependent.

Syntax

```
typedef struct RvSelectEngineInternal RvSelectEngine;
```

RvSelectFd

The file descriptor type. This structure holds information about a single file descriptor or socket that is waiting for events on a select engine. The contents are interface-dependent.

SELECT CALLBACKS

RvSelectCb()

DESCRIPTION

Callback that is executed when an event occurs on a file descriptor.

SYNTAX

```
typedef void(* RvSelectCb)(
  IN RvSelectEngine*    selectEngine,
  IN RvSelectFd*        fd,
  IN RvSelectEvents     selectEvent,
  IN RvBool             error);
```

PARAMETERS

selectEngine

The events engine that monitors this file descriptor.

fd

The file descriptor on which this event occurred.

selectEvent

The event that happened.

error

RV_TRUE if an error occurred.

REMARKS

- RvSelectRead is received whenever messages are waiting on the given connection.
- Applications must be prepared to get WOULDBLOCK events when actually trying to read from the connection.
- Multiple recv() calls are possible. The event will only be activated again once this callback returns.

- RvSelectWrite is received when the application can send messages on the connection without being blocked (again, they must be prepared to get WOULDBLOCK).
- After RvSelectWrite is received, applications are encouraged to remove this event and enable it again when getting blocked upon trying to send messages. This works both on UNIX and Windows systems.

RvSelectPreemptionCb()

DESCRIPTION

Callback that is executed when a preemption event, other than *RvSelectEmptyPreemptMsg* occurs. *RvSelectEmptyPreemptMsg* is defined as: #define RvSelectEmptyPreemptMsg ((RvUint8)0).

SYNTAX

```
typedef void
(* RvSelectPreemptionCb)(
  IN RvSelectEngine    *selectEngine,
  IN RvUint8           preemptEv,
  IN void              *context);
```

PARAMETERS

selectEngine

The preempted select engine.

preemptEv

Preemption event.

context

User specific preemption context.

RvTimerFunc()

DESCRIPTION

This function should be called when a timer is triggered.

SYNTAX

```
typedef RvBool (*RvTimerFunc)(void *userData)
```

PARAMETERS

userData

User parameter that was passed in RvSelectSetTimeoutInfo.

SELECT FUNCTIONS

RvSelectSetMaxFileDescriptors()

DESCRIPTION

Sets the amount of file descriptors that the Select module can handle in a single select engine. In most operating systems, this is also the value of the highest file descriptor possible.

SYNTAX

```
RvStatus RvSelectSetMaxFileDescriptors(
  IN RvUint32 maxFileDescriptors);
```

PARAMETERS

maxFileDescriptors

The maximum value of file descriptor that is possible.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- Check if your Stack and application uses this function before porting it.
- This function should be called before RvSelectConstruct() is called, and after RvSelectInit() is called.

RvSelectGetMaxFileDescriptors()

DESCRIPTION

Gets the current value of the maximum number of file descriptors that a single select engine can monitor.

SYNTAX

```
RvUint32 RvSelectGetMaxFileDescriptors(void);
```

PARAMETERS

None.

RETURN VALUES

Returns the maximum value for a file descriptor that select engines will support.

SELECT ENGINE FUNCTIONS

These functions create and handle the select engine itself. They use a select engine object.

RvSelectConstruct()

DESCRIPTION

Creates a new select engine.

SYNTAX

```
RvStatus RvSelectConstruct(
  IN  RvUint32          maxHashSize,
  IN  RvUint32          maxTimers,
  IN  RvLogMgr          *logMgr,
  OUT RvSelectEngine    **selectEngine);
```

PARAMETERS

maxHashSize

The hash size used by the engine.

Note This value does not indicate the maximum number of file descriptors, but rather the length of the internal hash table to be allocated. In this table each entry consists of a linked list of file descriptors.

maxTimers

The maximum number of timers that can be managed by the timer queue, initiated by the select engine.

logMgr

The log manager instance, or NULL.

selectEngine

The events engine to be created.

Note The select engine memory will be allocated in the default memory region.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSelectDestruct()

DESCRIPTION

Destructs a events engine.

If the select engine is being used by other Stack(s), it will not be destructed. In this case, the maximum possible number of timeouts that the select engine's timer queue can contain will be decreased. Only if a select engine is not being used by other Stacks, the select engine will be destructed and its memory will be freed.

SYNTAX

```
RvStatus RvSelectDestruct(
  IN RvSelectEngine*    selectEngine
  IN RvUint32           maxTimers);
```

PARAMETERS

selectEngine

The events engine to be destructed.

maxTimers

The maximum number of timers used when the select engine was constructed.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvSelectWaitAndBlock()

DESCRIPTION

Waits for events to occur on this engine and blocks the current running thread for a given amount of time, or until events occur.

SYNTAX

```
RvStatus RvSelectWaitAndBlock(
  IN RvSelectEngine*    selectEngine,
  IN RvUint64           nsecTimeout);
```

PARAMETERS

selectEngine

The events engine for which to wait.

nsecTimeout

The timeout to wait in nanoseconds. The RvSelectWaitAndBlock calculates the actual timeout to wait by choosing the closest value between the timer queue timeout and the nsecTimeout.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- This function must be called from the thread that has called RvSelectConstruct().
- The timeout parameter of this function is used by some of the Stacks to implement their timer mechanism.

SELECT PREEMPTION FUNCTIONS

The Select module implements the Preemption mechanism for the Wait and Block functionality (see RvSelectWaitAndBlock()). The Preemption mechanism is used mostly in a multi-threaded environment for sending requests to a thread that calls to the RvSelectWaitAndBlock() API. The preemption mechanism is used by the core of the Stack to notify about changes in the list of file descriptors or the timeout value.

RvSelectStopWaiting

DESCRIPTION

Stops waiting for events on the given engine.

SYNTAX

```
RvStatus RvSelectStopWaiting(
  IN RvSelectEngine    *selectEngine,
  IN RvUint8           message
  IN RvLogMgr          *logMgr);
```

PARAMETERS

selectEngine

The engine to stop from blocking.

message

The preemption message. Can be used for Stack-specific messages.

logMgr

The log manager. NULL value can be used.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

- This function causes calls to RvSelectWaitAndBlock() to return.
- This function calls the engine's preemptionCb callback if the message is not equal to the RvSelectEmptyPreemptMsg and if preemptionCb of the engine has been defined.

Select File Descriptor Functions

As stated earlier, sockets are the file descriptors in non-blocking mode that are waiting for events on them using a select engine. To create these file descriptors and register for events on them, you need the following functions:

RvFdConstruct()

DESCRIPTION

Constructs a file descriptor for a given socket. This function must not be applied on an fd structure after it was added to the RvSelect waiting loop (or before it was removed from it).

SYNTAX

```
RvStatus RvFdConstruct(
  IN RvSelectFd*    fd,
  IN RvSocket*      s
  IN RvLogMgr       *logMgr
```

PARAMETERS

fd

The file descriptor to construct.

s

The socket to use for this file descriptor.

logMgr

The log manager. NULL value can be used.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvFdDestruct()

DESCRIPTION

Destructs the file descriptor.

SYNTAX

```
RvStatus RvFdDestruct(
  IN RvSelectFd* fd);
```

PARAMETERS

fd

The file descriptor to be destructed.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

It is assumed that this function will not be called if the given socket is still monitoring events by a select engine for this file descriptor. RvSelectRemove() has been called for this file descriptor before this function is called.

RvFdGetSocket()

DESCRIPTION

Gets the socket associated with the file descriptor structure.

SYNTAX

```
RvSocket* RvFdGetSocket(
  IN RvSelectFd* fd);
```

PARAMETERS

fd

The file descriptor.

RETURN VALUES

Returns the socket associated with the file descriptor on success, NULL on failure.

RvSelectGetEvents()

DESCRIPTION

Returns the list of events being monitored for a particular file descriptor.

SYNTAX

```
RvSelectEvents RvSelectGetEvents(
  IN RvSelectFd* fd);
```

PARAMETERS

fd

The file descriptor.

RETURN VALUES

Returns the events being waited for.

RvSelectFindFd()

DESCRIPTION

Finds the file descriptor (RvSelectFd) by the socket associated with the file descriptor.

SYNTAX

```
RvSelectFd* RvSelectFindFd(
  IN RvSelectEngine*    selectEngine,
  IN RvSocket*          s);
```

PARAMETERS

selectEngine

The events engine in which to look.

s

The socket/file descriptor object to find.

RETURN VALUES

Returns the RvSelectFd object if one exists, NULL otherwise.

REMARKS

Note that a problem might occur if you rely on the output of the function. After the function is called, the list of the file descriptor structure may be changed by another thread.

RvSelectAdd()

DESCRIPTION

Adds a new file descriptor to a list of file descriptors to be monitored by this select engine.

SYNTAX

```
RvStatus RvSelectAdd(
  IN RvSelectEngine*    selectEngine,
  IN RvSelectFd*        fd,
  IN RvSelectEvents     selectEvents,
  IN RvSelectCb         eventsCb);
```

PARAMETERS

selectEngine

The events engine of this file descriptor.

fd

The file descriptor to check.

selectEvents

The events to check.

eventsCb

The callback to use when these events occur.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

A file descriptor cannot be added twice. Once added, only RvSelectUpdate() can be called to update the events being waiting on, or RvSelectRemove() can be called to remove this file descriptor from being handled by this select engine.

RvSelectRemove()

DESCRIPTION

Removes a file descriptor from a list of file descriptors that are being monitored by this select engine.

SYNTAX

```
RvStatus RvSelectRemove(
  IN RvSelectEngine*    selectEngine,
  IN RvSelectFd*        fd);
```

PARAMETERS

selectEngine

The events engine of this file descriptior.

fd

The file descriptor to remove.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

This function should only be called for file descriptors for which RvSelectAdd() was called.

RvSelectUpdate()

DESCRIPTION

Updates the list of events that are monitored by the select engine for the file descriptor.

SYNTAX

```
RvStatus RvSelectUpdate(
  IN RvSelectEngine*    selectEngine,
  IN RvSelectFd*        fd,
  IN RvSelectEvents     selectEvents,
  IN RvSelectCb         eventsCb);
```

PARAMETERS

selectEngine

The events engine of this file descriptor.

fd

The file descriptor to update.

selectEvents

Events to check.

eventsCb

The callback to use when these events occur.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

REMARKS

This function should only be called for file descriptors for which RvSelectAdd() was called.

Select Timer Management Functions

RvSelectSetTimeoutInfo()

Description

Sets Stack timeout processing callback. Useful when the Stack wants to process timeouts differently than by simply applying the timer service routine. Thread-safe functions under assumptions specified in the header of the file.

Syntax

```
RvStatus RVCALLCONV RvSelectSetTimeoutInfo(
  IN RvSelectEngine    *selectEngine,
  IN RvTimerFunc       timeOutCb,
  IN void              *cbContext);
```

Parameters

selectEngine

Select engine object.

timeOutCb

Timeout callback.

cbContext

Callback user data.

Return Values

RV_OK on success, other on failure.

RvSelectGetTimeoutInfo()

DESCRIPTION

Retrieves Stack timeout processing information, including the callback and timeout events queue. Thread-safe functions under assumptions specified in the header of the file.

SYNTAX

```
RvStatus RVCALLCONV RvSelectGetTimeoutInfo(
  IN  RvSelectEngine    *selectEngine,
  OUT RvTimerFunc       *timeOutCb,
  OUT RvTimerQueue      **tqueue);
```

PARAMETERS

selectEngine

Select engine object.

timeOutCb

Timeout callback.

tqueue

Timeout events queue.

RETURN VALUES

RV_OK on success, other on failure.

HOST MODULE

The Host module is for getting the name and IP list of the local host being used. If such information cannot be obtained by your operating system, you must implement other means of doing so (such as checking environment variables or using static lists).

You can find this module in *ccore/rvhost.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- This module is only used for accessing local host information.

HOST FUNCTIONS

RvHostLocalGetName()

DESCRIPTION

Gets the name of the local host.

SYNTAX

```
RvStatus RvHostLocalGetName(
  IN  RvSize_t     nameLength,
  IN  RvLogMgr*    logMgr,
  OUT RvChar*      name);
```

PARAMETERS

nameLength

The length of the name buffer in bytes.

logMgr

The log manager instance, or NULL.

name

The null terminated name of the local host.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvHostLocalGetAddress()

DESCRIPTION

Gets the addresses of the local host.

SYNTAX

```
RvStatus RvHostLocalGetAddress(
  IN    RvLogMgr*     logMgr,
  INOUT RvUint32*     numberOfAddresses,
  INOUT RvAddress*    addresses);
```

PARAMETERS

logMgr

The log manager instance, or NULL.

numberOfAddresses

The maximum number of addresses to get.

addresses

The array of addresses to fill in.

NumberOfAddresses

The number of addresses filled in.

addresses

The constructed addresses. These addresses should be destructed using RvAddressDestruct() when not needed anymore.

RETURN VALUES

Returns RV_OK on success, other on failure.

TLS MODULE

The TLS module provides the TLS (Transport Layer Security) interface. The TLS module assumes that OpenSSL version 0.9.6 or 0.9.7 is installed (see http://www.openssl.org for the OpenSSL specific information).

The functions listed in this section should be implemented in case your Stack needs this module to be ported and you are unable to install OpenSSL.

You can find this module in *ccore/netdrivers/rvsocket.c ccore/netdrivers/rvtls.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

None.

TLS Constants and Type Definitions

Certificate Comparison Callback

RvCompareCertificateCB()

Description

Callback that is executed when a peer certificate will be received.

Syntax

```
typedef RvInt ( * RvCompareCertificateCB ) (
  IN int     prevErrro,
  IN void    *certCtx);
```

Parameters

prevErrro

The error that was encountered during comparison of previous certificates for the same TLS session.

certCtx

The certificate.

SSL METHOD

RvTLSMethod

Defines supported SSL versions. Following SSL versions are supported:

- SSL_V1—SSL version 1
- SSL_V2—SSL version 2
- TLS_V1—SSL version 3

SYNTAX

```
typedef enum {
  RV_TLS_SSL_V2 = 1,
  RV_TLS_SSL_V3,
  RV_TLS_TLS_V1
} RvTLSMethod;
```

DEFAULT CERTIFICATE CHAIN LENGTH

RV_TLS_DEFAULT_CERT_DEPTH

Defines the default value of the maximum possible certificate chain length. By default, certificate chain length is not constrained.

SYNTAX

```
#define RV_TLS_DEFAULT_CERT_DEPTH    (-1)
```

TLS EVENT

RvTLSEvents

TLS-specific events that may be received by the application when using the TLS module. The following TLS events may be encountered:

- TLS handshake event—Notifies that the TLS handshake procedure may be continued.
- TLS read event—Notifies that TLS read may be applied.
- TLS write event—Notifies that TLS write may be applied.
- TLS shutdown event—Notifies that TLS shutdown procedure can be applied.

SYNTAX

```
typedef RvUint16 RvTLSEvents;

#define RV_TLS_HANDSHAKE_EV 0x1
#define RV_TLS_READ_EV 0x2
#define RV_TLS_WRITE_EV 0x4
#define RV_TLS_SHUTDOWN_EV 0x8
```

SUPPORTED PRIVATE KEY TYPES

RvPrivKeyType

Defines the supported private key types. Currently only the RSA key type is supported.

SYNTAX

```
typedef enum {
  RV_TLS_RSA_KEY = 1
} RvPrivKeyType;
```

TLS ENGINE

RvTLSEngine

Defines the TLS engine. The Stack may define multiple TLS engines, used to manage TLS sessions. TLS sessions created on the same TLS engine share following initial parameters:

- SSL version (can be SSL v1, SSL v2 or TLS v1)
- Private key
- Private key type
- Private key length
- Local side certificate
- Local side certificate length
- Maximum allowed depth of certificates

SYNTAX

```
typedef struct {
  SSL_CTX *ctx;
} RvTLSEngine;
```

TLS SESSION

RvTLSSession

Defines the TLS session. The TLS session is opened over the TCP connection and is used to initiate the TLS connection and for transmitting or receiving TLS data.

SYNTAX

```
typedef struct {
  SSL *sslSession;
  BIO *bio;
  RvSelectEvents requiredForHandshake;
  RvSelectEvents requiredForTLSRead;
  RvSelectEvents requiredForTLSWrite;
  RvSelectEvents requiredForTLSShutdown;
  RvTLSEvents tlsEvents;
} RvTLSSession;
```

TLS FUNCTIONS

RvTLSEngineConstruct()

DESCRIPTION

Creates and initiates the TLS module engine.

SYNTAX

```
RvStatus RvTLSEngineConstruct(
  IN  RvTLSMethod      method,
  IN  RvChar           *privKey,
  IN  RvPrivKeyType    privKeyType,
  IN  RvInt            privKeyLen,
  IN  RvChar           *cert,
  IN  RvInt            certLen,
  IN  RvInt            certDepth,
  IN  RvMutex          *mtx,
  IN  RvLogMgr         *logMgr,
  OUT RvTLSEngine      *tlsEngine);
```

PARAMETERS

method

SSL version. Can be SSL v1, SSL v2 or TLS v1.

privKey

Private key.

privKeyType

Private key type.

privKeyLen

Private key length.

cert

Local side certificate.

certLen

Local side certificate length.

certDepth

Maximum allowed depth of certificates.

mtx

Mutex that protects this TLS.

logMgr

Log instance, or NULL.

tlsEngine

Constructed TLS engine. The user should allocate memory for the tlsEngine before calling RvTLSEngineConstruct.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSEngineAddAutorityCertificate()

DESCRIPTION

Adds certificate authority to the TLS engine.

SYNTAX

```
RvStatus RvTLSEngineAddAutorityCertificate(
  IN RvTLSEngine    *tlsEngine,
  IN RvChar         *cert,
  IN RvInt          certLen,
  IN RvMutex        *mtx,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

tlsEngine

The TLS engine where the certificate will be added.

cert

Certificate authority (CA) certificate.

certLen

Certificate authority (CA) certificate length.

mtx

Mutex that protects this TLS engine memory.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSEngineAddCertificate()

DESCRIPTION

Adds a certificate to the engine certificates chain.

SYNTAX

```
RvStatus RvTLSEngineAddCertificate(
  IN RvTLSEngine    *tlsEngine,
  IN RvChar         *cert,
  IN RvInt          certLen,
  IN RvMutex        *mtx,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

tlsEngine

TLS engine where certificate will be added.

cert

Local side certificate.

certLen

Local side certificate length.

mtx

Mutex that protects this TLS engine.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSEngineDestruct()

DESCRIPTION

Destructs the TLS engine.

SYNTAX

```
RvStatus RvTLSEngineDestruct(
  IN RvTLSEngine    *tlsEngine,
  IN RvMutex        *mtx,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

tlsEngine

Engine to be destructed.

mtx

Mutex that protects this TLS engine.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSSessionConstruct()

DESCRIPTION

Creates an uninitialized SSL/TLS session.

SYNTAX

```
RvStatus RvTLSSessionConstruct(
  IN  RvTLSEngine     *tlsEngine,
  IN  RvMutex         *mtxEngine,
  IN  RvMutex         *mtxSession,
  IN  RvLogMgr        *logMgr,
  OUT RvTLSSession    *tlsSession);
```

PARAMETERS

tlsEngine

TLS engine.

mtxEngine

Mutex that protects the TLS engine.

mtxSession

Mutex that protects the TLS session engine.

logMgr

Log instance, or NULL.

tlsSession

New TLS session.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSSessionDestruct()

DESCRIPTION

Destructs the TLS session.

SYNTAX

```
RvStatus RvTLSSessionDestruct(
  IN RvTLSSession    *tlsSession,
  IN RvMutex         *mtx,
  IN RvLogMgr        *logMgr);
```

PARAMETERS

tlsSession

The TLS session to be destructed.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSSessionClientHandshake()

DESCRIPTION

Initiates client connection to a remote TLS server.

SYNTAX

```
RvStatus RvTLSSessionClientHandshake(
  IN RvTLSSession              *tlsSession,
  IN RvCompareCertificateCB    certCB,
  IN RvSocket                  tcpSock,
  IN RvMutex                   *mtx,
  IN RvLogMgr                  *logMgr);
```

PARAMETERS

tlsSession

The TLS session that had already been constructed.

certCB

Certificate callback. If it is not NULL, it enables a certificate check by the client.

tcpSock

TCP connection socket.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSSessionServerHandshake()

DESCRIPTION

Server-side SSL/TLS handshake. Responses to client handshake request.

SYNTAX

```
RvStatus RvTLSSessionServerHandshake(
  IN RvTLSSession              *tlsSession,
  IN RvCompareCertificateCB    certCB,
  IN RvSocket                  tcpSock,
  IN RvMutex                   *mtx,
  IN RvLogMgr                  *logMgr);
```

PARAMETERS

tlsSession

The TLS session that had already been created.

certCB

Certification callback. If it is not NULL, the server will request client certification.

tcpSock

TCP connection socket.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSSessionReceiveBuffer()

DESCRIPTION

Reads and decrypts the TLS message. Saves the retrieved message or part of it into the receiveBuf buffer.

SYNTAX

```
RvStatus RvTLSSessionReceiveBuffer(
  IN    RvTLSSession    *tlsSession,
  IN    RvChar          *receiveBuf,
  IN    RvMutex         *mtx,
  IN    RvLogMgr        *logMgr,
  INOUT RvInt           *receiveBufLen);
```

PARAMETERS

tlsSession

The TLS session.

receiveBuf

Buffer for received data.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

receiveBufLen

In input—Receive buffer maximum length.

In output—Length of received data.

RETURN VALUES

Returns RV_OK on success, a ‘would block’ error when the operation remains incomplete because of a non-blocking TCP socket, or a general failure error.

RvTLSSessionSendBuffer()

DESCRIPTION

Encrypts the Stack message and sends the TLS message with encrypted data to the TLS session peer.

SYNTAX

```
RvStatus RvTLSSessionSendBuffer(
  IN RvTLSSession    *tlsSession,
  IN RvChar          *sendBuf,
  IN RvInt           sendBufLen,
  IN RvMutex         *mtx,
  IN RvLogMgr        *logMgr);
```

PARAMETERS

tlsSession

Connected TLS session.

sendBuf

Buffer to send.

sendBufLen

Length of data in the sendBuf.

mtx

Mutex that protects the TLS session.

logMgr

Log instance or NULL.

RETURN VALUES

Returns RV_OK on success, a ‘would block’ error when the operation remains incomplete because of a non-blocking TCP socket, or a general failure error.

RvTLSSessionShutdown()

DESCRIPTION

Sends a Shutdown request to the SSL/TLS connection peer.

SYNTAX

```
RvStatus RvTLSSessionShutdown(
  IN RvTLSSession    *tlsSession,
  IN RvMutex         *mtx,
  IN RvLogMgr        *logMgr);
```

PARAMETERS

tlsSession

Connected TLS session.

mtx

Mutex that protects the session structure.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, a ‘would block’ error when the operation remains incomplete because of a non-blocking TCP socket, an ‘incomplete’ error when shutdown remains incomplete because of the peer response is lacking, or a general failure error.

RvTLSTranslateSelectEvents()

DESCRIPTION

Translates received Select module events to the appropriate TLS event.

SYNTAX

```
RvStatus RvTLSTranslateSelectEvents(
  IN  RvTLSSession      *tlsSession,
  IN  RvSelectEvents    selEvents,
  IN  RvMutex           *mtx,
  IN  RvLogMgr          *logMgr,
  OUT RvTLSEvents       *tlsEvents);
```

PARAMETERS

tlsSession

Connected TLS session.

selEvents

Select module events to be translated.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

tlsEvents

Translated TLS events.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSTranslateTLSEvents()

DESCRIPTION

Translates received TLS events to the appropriate Select module event.

SYNTAX

```
RvStatus RvTLSTranslateTLSEvents(
  IN  RvTLSSession      *tlsSession,
  IN  RvTLSEvents       tlsEvents,
  IN  RvMutex           *mtx,
  IN  RvLogMgr          *logMgr,
  OUT RvSelectEvents    *selEvents);
```

PARAMETERS

tlsSession

Connected TLS session.

tlsEvents

TLS event.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

selEvents

Translated Select module events.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSGetCertificateLength()

DESCRIPTION

Retrieves the length of the ASN1-formatted certificate. Normally used by a certificate verification callback on the certificate context specified in the callback input. This function is used by the Stack to learn how much memory should be allocated for the certificate.

SYNTAX

```
RvStatus RvTLSGetCertificateLength(
  IN  void     *certCtx,
  OUT RvInt    *certLen);
```

PARAMETERS

certCtx

Certificate context.

certLen

Length of the ASN1 certificate.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSGetCertificate()

DESCRIPTION

Retrieves the ASN1-format certificate from a certificate context input of the certificate verification callback.

SYNTAX

```
RvStatus RvTLSGetCertificate(
  IN  void      *certCtx,
  OUT RvChar    *cert);
```

PARAMETERS

certCtx

Certificate context.

cert

Retrieved certificate.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

The user should provide allocated memory space big enough for keeping the ASN1 certificate.

RvTLSSessionGetCertificateLength()

DESCRIPTION

Retrieves the ASN1-format certificate length from a session that completes the handshake. This function is used to learn how much memory should be allocated for the certificate.

SYNTAX

```
RvStatus RvTLSSessionGetCertificateLength(
  IN  RvTLSSession    *tlsSession,
  IN  RvMutex         *mtx,
  IN  RvLogMgr        *logMgr,
  OUT RvInt           *certLen);
```

PARAMETERS

tlsSession

Connected TLS session.

mtx

Mutex that protects the TLS session.

logMgr

Log instance, or NULL.

certLen

Length of ASN1 certificate.

RETURN VALUES

Returns RV_OK on success, other on failure.

RvTLSSessionGetCertificate()

DESCRIPTION

Retrieves the ASN1-format certificate from a session after the handshake is complete.

SYNTAX

```
RvStatus RvTLSSessionGetCertificate(
  IN  RvTLSSession    *tlsSession,
  IN  RvMutex         *mtx,
  IN  RvLogMgr        *logMgr,
  OUT RvChar          *cert);
```

PARAMETERS

tlsSession

Connected TLS session.

mtx

Mutex that protects session structure.

logMgr

Log instance, or NULL.

cert

Retrieved certificate.

RETURN VALUES

Returns RV_OK on success, other on failure.

REMARKS

The user should provide allocated memory space big enough for keeping the ASN1 certificate.

RvTLSGetCertificateVerificationError()

DESCRIPTION

Retrieves a certificate verification error as text.

SYNTAX

```
void RvTLSGetCertificateVerificationError(
  IN  void      *cert,
  OUT RvChar    **errString);
```

PARAMETERS

cert

Certificate context, received by the verification callback.

errString

Retrieved error string.

RETURN VALUES

N/A

RvTLSSessionCheckCertAgainstName()

DESCRIPTION

Validates the correctness of the certificate by comparing the values saved under the DNS key in the certificate name.

SYNTAX

```
RvStatus RvTLSSessionCheckCertAgainstName(
  IN RvTLSSession    *tlsSession,
  IN RvChar          *name,
  IN RvMutex         *mtx,
  IN RvChar          *memBuf,
  IN RvInt32         memBufLen,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

tlsSession

Connected TLS session.

name

Domain name string.

mtx

Mutex that protects the TLS session.

memBuf

Memory buffer used to save intermediate data during processing. Used to ensure thread-safety of the function.

memBufLen

Size of the memBuf.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK if certificate matches the name, failure in case of error or if the certificate does not match the name.

RvTLSEngineCheckPrivateKey()

DESCRIPTION

Checks the consistency of a private key with the corresponding certificate loaded into an engine.

SYNTAX

```
RvStatus RvTLSEngineCheckPrivateKey(
  IN RvTLSEngine    *tlsEngine,
  IN RvMutex        *mtx,
  IN RvLogMgr       *logMgr);
```

PARAMETERS

tlsEngine

TLS engine where the certificate is to be added.

mtx

Mutex that protects this TLS engine.

logMgr

Log instance, or NULL.

RETURN VALUES

Returns RV_OK on success, other on failure.

6

UTILITIES AND STANDARD ANSI

These modules allow easy access to functions and implementations that can be considered as ANSI C or close to ANSI C. They deal with the fact that ANSI C is implemented quite differently in different operating systems, and some operating systems and compilers do not even fully support ANSI C.

Before you start porting these modules, please make sure that the functions you are porting are actually needed by the Stack you are using and your application.

The functions, constants and type definitions included in this chapter are:

64ASCII MODULE

64Ascii Constants and Type Definitions

- RV_64TOASCII_BUFSIZE
- RV_64TOHEX_BUFSIZE

64Ascii Functions

- Rv64UtoA()
- Rv64toA()
- Rv64UtoHex()

ASSERT MODULE

Assert Functions

- RvAssert()

STDIO MODULE

LOGLISTENER

Log Listener Constants and Type Definitions

- RvLogListener

Log Listener Functions

- RvLogListenerConstructTerminal()
- RvLogListenerDestructTerminal()
- RvLogListenerConstructLogfile()
- RvLogListenerLogfileGetCurrentFilename()
- RvLogListenerDestructLogfile()
- RvLogListenerConstructDebug()
- RvLogListenerDestructDebug()

64BITS MODULE

64ASCII MODULE

The 64Ascii module provides functions for converting 64 bit numbers to strings. The module is mostly used for logging purposes. A standard, manual implementation is supplied that should work on all operating systems, even those with poor support of 64 bit numbers conversions.

You can find this module in *ccore/rv64ascii.c*.

ASSUMPTIONS TO YOU NEED TO FOLLOW

- If you are not sure if you have good conversion facilities for 64 bit numbers to strings, use the supplied RV_64TOASCII_MANUAL interface or try to see if the RV_64TOASCII_STANDARD interface works on your operating system.

64ASCII CONSTANTS AND TYPE DEFINITIONS

RV_64TOASCII_BUFSIZE

The minimum buffer size required for ASCII strings of 64 bit numbers.

RV_64TOHEX_BUFSIZE

The minimum buffer size required for HEX strings of 64 bit numbers.

64ASCII FUNCTIONS

Rv64UtoA()

DESCRIPTION

Converts an unsigned 64 bit integer (RvUint64) to a decimal string.

SYNTAX

```
RvChar *Rv64UtoA(
  IN  RvUint64    num64,
  OUT RvChar      *buf);
```

PARAMETERS

buf

The buffer where resulting string will be placed.

num64

The unsigned 64 bit number to be converted.

RETURN VALUES

Returns a pointer to *buf* if successful, otherwise NULL.

REMARKS

The buffer (buf) should be at least RV_64TOASCII_BUFSIZE bytes long or the end of the buffer may be overwritten.

Rv64toA()

DESCRIPTION

Converts a signed 64 bit integer (RvInt64) to a decimal string.

SYNTAX

```
RvChar *Rv64toA(
  IN  RvInt64    num64,
  OUT RvChar     *buf);
```

PARAMETERS

buf

The buffer where resulting string will be placed.

num64

The signed 64 bit number to be converted.

RETURN VALUES

Returns a pointer to *buf* if successful, otherwise NULL.

REMARKS

The buffer (buf) should be at least RV_64TOASCII_BUFSIZE bytes long or the end of the buffer may be overwritten.

Rv64UtoHex()

DESCRIPTION

Converts an unsigned 64 bit integer (RvUint64) to a hexadecimal string.

SYNTAX

```
RvChar *Rv64UtoHex(
  IN  RvUint64    num64,
  OUT RvChar      *buf);
```

PARAMETERS

buf

The buffer where resulting string will be placed.

num64

The unsigned 64 bit number to be converted.

RETURN VALUES

Returns a pointer to *buf* if successful, otherwise NULL.

REMARKS

The buffer (buf) should be at least RV_64TOHEX_BUFSIZE bytes long or the end of the buffer may be overwritten.

ASSERT MODULE

The Assert module replaces the ANSI C *assert.h* for portability. The functions are a one-to-one mapping with the ANSI C functions in *assert.h*.

You can find this module in *ccore/rvassert.h*, by changing the definition of RV_ASSERT_ENABLE.

ASSUMPTIONS YOU NEED TO FOLLOW

- If you do not want to use assertions at all while debugging, or if your compiler does not support assertions, you can disable them in *ccore/rvccoreconfig.h*.

ASSERT FUNCTIONS

RvAssert()

DESCRIPTION

This function is equivalent to the ANSI C assert macro. If the test fails, the macro will try to display an error message and then shut down the system.

SYNTAX

```
void RvAssert(
  RvBool test);
```

PARAMETERS

test

The expression to assert.

RETURN VALUES

None.

REMARKS

By default, this function is not compiled if RV_DEBUG is not defined.

StdIo Module

This module contains ports for standard I/O functions that require porting based on an operating system or compiler.

The list of functions implemented in this module varies between operating systems and compilers due to the differences between them. This is where you should implement all the functions not supported by your operating system of choice. (If you look at the current implementation of StdIo module for Windows CE 2.11, you can see the implementation of the toupper() function, for example).

You can find this module in *ccore/rvstdio.c*.

Assumptions You Need To Follow

- You do not have to port all the functions that handle files for operating systems without any file system. The file system is mainly used by the Stacks to read configuration files automatically and write log files.
- Several functions for manipulation of 64 bit numeric values are also implemented here. You can write your own operating system-specific implementation or use the regular macros if possible.
- Most of the implementations are macro definitions to the corresponding ANSI C function. (These functions are not documented because they are almost exact copies of the ANSI C functions.)

LOGLISTENER

The LogListener module is responsible for handling log messages and printing them out to standard output, file or other facilities. The functions are implemented here as logging facilities that can be used during the development phase of your application.

You can skip porting these functions, since they are used for logging purposes only. You can supply your own logging function to the Stack you are using.

You can find this module in *ccore/rvloglistener.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

- None.

LOG LISTENER CONSTANTS AND TYPE DEFINITIONS

RvLogListener

Holds information about the listeners supported by the Core. This includes only the type of the listener.

Only one type of each listener can be constructed—this is only the case for the listeners that the Core implements. Applications can write their own listeners and use them as desired.

Syntax

```
typedef struct{
  RvLogMgr* logMgr;
  int listenerType;
}RvLogListener;
```

Parameters

logMgr

The Log Manager this listener uses.

listenerType

The type of listener that is constructed.

LOG LISTENER FUNCTIONS

RvLogListenerConstructTerminal()

DESCRIPTION

Constructs a log listener that sends log messages to the terminal, using standard output or standard error.

SYNTAX

```
RvStatus RvLogListenerConstructTerminal(
  IN RvLogListener*    listener,
  IN RvLogMgr*         logMgr,
  IN RvBool            stdOut);
```

PARAMETERS

listener

The listener to construct.

logMgr

The Log Manager to which to listen.

stdOut

RV_TRUE for stdout, RV_FALSE for stderr.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

RvLogListenerDestructTerminal()

DESCRIPTION

Destructs the terminal listener.

SYNTAX

```
RvStatus RvLogListenerDestructTerminal(
  IN RvLogListener* listener);
```

PARAMETERS

listener

The listener to destruct.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

RvLogListenerConstructLogfile()

DESCRIPTION

Constructs a log listener that sends log messages to a file.

SYNTAX

```
RvStatus RvLogListenerConstructLogfile(
  IN RvLogListener*    listener,
  IN RvLogMgr*         logMgr,
  IN const RvChar*     fileName,
  IN RvUint32          numFiles,
  IN RvUint32          fileSize,
  IN RvBool            flushEachMessage);
```

PARAMETERS

listener

The listener to construct.

logMgr

The Log Manager to which to listen.

fileName

The name of the log file.

numFiles

The number of cyclic files to use.

fileSize

The size of each file in cycle in bytes. This parameter is only applicable if *numFiles* >1.

flushEachMessage

RV_TRUE if each message written to the logfile is to be flushed.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

RvLogListenerLogfileGetCurrentFilename()

DESCRIPTION

Gets the filename of the current file being written.

SYNTAX

```
RvStatus RvLogListenerLogfileGetCurrentFilename(
  IN  RvLogListener*    listener,
  IN  RvUint32          fileNameLength,
  OUT RvChar*           fileName);
```

PARAMETERS

listener

The listener to check.

fileNameLength

The maximum length of the filename.

fileName

The filename of the current file being written.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

RvLogListenerDestructLogfile()

DESCRIPTION

Destructs the log file listener.

SYNTAX

```
RvStatus RvLogListenerDestructLogfile(
  IN RvLogListener* listener);
```

PARAMETERS

listener

The listener to destruct.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

RvLogListenerConstructDebug()

DESCRIPTION

Constructs a log listener that sends log messages to the debug window of Visual C.

SYNTAX

```
RvStatus RvLogListenerConstructDebug(
  IN RvLogListener*    listener,
  IN RvLogMgr*         logMgr);
```

PARAMETERS

listener

The listener to construct.

logMgr

The Log Manager to which to listen.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

REMARKS

This function is only applicable for Win32 applications.

RvLogListenerDestructDebug()

DESCRIPTION

Destructs the debug listener.

SYNTAX

```
RvStatus RvLogListenerDestructDebug(
  IN RvLogListener* listener);
```

PARAMETERS

listener

The listener to destruct.

RETURN VALUES

Returns RV_OK on success, or a negative value on failure.

64BITS MODULE

This module provides functions for arithmetic and logic operations of 64-bit numbers, which can be used in applications that are compiled by tools that do not supply a built-in support.

Two configuration options are provided:

- Standard—Used for compilers which do have a built-in support. In this case, all functions are defined as simple macros that use the standard arithmetic symbols.
- Manual—Used where no built-in support exists. In this case, the functions provide 16-bit based implementation, which is somewhat slower than the standard configuration but allows the use of 64-bit arithmetic operations.

You can find this module in *cutils/rv64bits.c*.

ASSUMPTIONS YOU NEED TO FOLLOW

None. You will probably not need to modify or add any new interfaces here. If you cannot compile or link your code due to 64-bit integer types, it will suffice to set the configuration from Standard (the default for most operating systems that are currently supported) to Manual.

7

UPDATING THE BUILD ENVIRONMENT

The Common Core uses several general definitions. Some of these definitions are compiler-dependant, which can be found in the C header file of the compiler. (For example, GNU GCC definitions are in *common/config/tool/rvgnu.h*).

These definitions are then used to define the general types in *common/cutils/rvtypes.h*.

REQUIRED DEFINITIONS

The required definitions are as follows:

`RV_SIZET_TYPE`	`size_t type (should always be size_t)`
`RV_PTRDIFFT_TYPE`	`ptrdiff_t type (should always be ptrdiff_t)`
`RV_CHAR_TYPE`	`standard character type (for char and char * only)`
`RV_VAR_INT_TYPE`	`standard variable (non-fixed) size signed variable`
`RV_VAR_UINT_TYPE`	`standard variable (non-fixed) size unsigned variable`
`RV_SIGNED_INT8_TYPE`	`8 bit signed variable`
`RV_UNSIGNED_INT8_TYPE`	`8 bit unsigned variable`
`RV_SIGNED_INT16_TYPE`	`16 bit signed variable`
`RV_UNSIGNED_INT16_TYPE`	`16 bit unsigned variable`
`RV_SIGNED_INT32_TYPE`	`32 bit signed variable`
`RV_UNSIGNED_INT32_TYPE`	`32 bit unsigned variable`
`RV_SIGNED_INT64_TYPE`	`64 bit signed variable`

RV_UNSIGNED_INT64_TYPE	64 bit unsigned variable
RV_VAR_INT_SUFFIX(n)	standard variable (non-fixed) size signed constant suffix
RV_VAR_UINT_SUFFIX(n)	standard variable (non-fixed) size unsigned constant suffix
RV_SIGNED_INT8_SUFFIX(n)	8 bit signed constant suffix
RV_UNSIGNED_INT8_SUFFIX(n)	8 bit unsigned constant suffix
RV_SIGNED_INT16_SUFFIX(n)	16 bit signed constant suffix
RV_UNSIGNED_INT16_SUFFIX(n)	16 bit unsigned constant suffix
RV_SIGNED_INT32_SUFFIX(n)	32 bit signed constant suffix
RV_UNSIGNED_INT32_SUFFIX(n)	32 bit unsigned constant suffix
RV_SIGNED_INT64_SUFFIX(n)	64 bit signed constant suffix
RV_UNSIGNED_INT64_SUFFIX(n)	64 bit unsigned constant suffix
RV_VAR_INT_MAX	maximum value of the standard variable (non-fixed) size signed variable
RV_VAR_INT_MIN	minimum value of the standard variable (non-fixed) size signed variable
RV_VAR_UINT_MAX	maximum value of the standard variable (non-fixed) size unsigned variable

RvStatus

Another definition, which should not be ported, but should be known is RvStatus. RvStatus is declared in *common/cutils/rverror.h* and contains the return results of most of the functions of the Common Core.

The success value for RvStatus is RV_OK, which is 0. Negative values indicate failure, while positive values indicate warnings.

Build Environment

The build environment is managed by GNU make. The version of GNU make required is 3.78 or higher.

If a Windows-based host is used for the porting process then a GNU Cygwin environment must be installed, including utilities like *sed, date, cp, mkdir* etc.

In the base directory of the Stack you are about to port is the main Makefile, which is used to compile the Stack, along with any other necessary modules. This file contains the main list of dependencies that will be processed.

Each component (a set of source files organized in a separate folder, also known as 'project') that is being compiled has its own *<component>.mak* file with instructions of how it should be compiled. Each such makefile also includes *common/config/make/project.mak*, which in turn creates the relevant file lists for each stage of the compilation process.

A set of additional makefiles that are operating system-dependent is found in the *common/config/make* directory. Only one of the makefiles in that directory will actually be used—the name of the makefile used is that of the operating system for which you are compiling (the target machine) with the prefix "os_".

A set of additional makefiles that are compiler-dependent is in the *common/config/make* directory. Only one of the makefiles in that directory will actually be used—the name of the makefile used is that of the compiler you will be using with the prefix "tool_".

Compiler-Related Files

Each compiler is defined in the Common Core by two files:

- The makefile used by GNU make. This file has the name of the compiler itself with the prefix "tool_". The GNU GCC compiler makefile, for example, is called *common/config/make/tool_gnu.mak*. This file holds all the compiler settings and the means to create the dependency lists.
- The C header file used to hold all compiler-dependent types. This file has the name of the compiler itself with an “rv” prefix. The GNU GCC compiler C header file, for example, is called *common/config/tool/rvgnu.h*.

Operating System-Related Files

Each operating system is defined in the Common Core by two files:

- The makefile used by GNU make. This file has the name of the operating system itself with the prefix "os_". The Solaris operating system makefile, for example, is called *common/config/make/os_solaris.mak*. This file holds all the operating system settings used during compilation.
- The C header file used to hold all operating system-dependent types. This file also defines which interface should be used in each of the Low Level modules of the Common Core. The file has the name of the operating system itself with an “rv” prefix. The Solaris operating system C header file, for example, is called *common/config/os/rvsolaris.h*.

Stack-Specific Build Environment Files

The Makefile automatically includes all files with the '.mak' extension from directory 'extensions'. These are used by the Stacks to add Stack-specific make parameters.

Porting Process

Porting the Low Level Functions of the Common Core is a process that works on several levels:

- Adding a new operating system to the list of supported operating systems, along with all of the configuration parameters that are specific to that operating system. It might also be updating the existing settings of an operating system currently supported, to allow the use of other compilers or different versions.
- Adding a new compiler to the list of supported compilers, along with all of the configuration parameters that are specific to that compiler. It might also be done just by adding settings to a compiler already supported, but to operating systems or compiler versions that are not currently supported by the Common Core.
- Adding new interfaces to existing modules to support the newly ported operating system. It might also be done just by modifying an existing implemented interface to support the ported operating system.

Adding a New Operating System

You are probably reading this document because you are using an operating system that is not currently supported by the Stack that you purchased from RADVISION. Even if you are just using an operating system already supported, but of a version that is not supported, you are encouraged to read this chapter.

Adding support for the Stack to the new operating system starts in this chapter. You first have to make sure the makefiles and related header files have your new operating system configured for them.

First of all, create the Operating System-Related Files. You can do this by copying files of another operating system that is close to yours, and then modifying them to fit your operating system. You can find these files in the *common/config/os* and *common/config/make* directories. Then do the following:

1. When writing the makefile for your operating system (*common/config/make/os_<os>.mak*), do not forget to set the

LEGAL_TARGET_OS_VERSION to the versions of the operating systems that you want to support.

Make sure you also set the list of operating system-dependent libraries needed to link executables (add them to the LIBS_LIST variable).

Also set the adapter folder list in the make variable ADAPTERS_$(TARGET_OS) as described in the section Adding a New Interface to a Module.

Most of the interesting parameters and settings will have to be added to the makefile of the compiler and not the makefile of the operating system.

2. When dealing with the header file for the operating system, do not forget to include any header file that must be included first (such as *vxworks.h* for the VxWorks operating system).

   At this stage, you should also set the interface used by each Low Level module. You can skip this stage until you get the environment working and then start this process, porting the modules one by one and setting their interfaces accordingly.

3. Add the new operating system to the list of supported operating systems. This list can be found in *common/config/os/rvosdefs.h*. Do not forget to define the specific operating system versions you intend to support.

4. Add this definition and the name as a string to the *common/ccore/rvccorestrings.h* file. This string can be fetched from within the application by calling the RvCCoreInterfaces() function to identify the configured operating system and to solve troubleshooting problems in the future.

5. Add an include directive to your newly created header file to *common/config/os/rvosconfig.h*.

6. In the file *common/config/make/environment.mak*, add the new operating system to the list of supported operating systems specified by the variable LEGAL_TARGET_OS.

7. Modify the *default.mak* file with the information of your new operating system. Do not forget to set the version as well as the operating system itself in this file.

You can now proceed to the settings of the compiler.

ADDING A NEW COMPILER

You probably need to follow the steps in this chapter even if you are only adding a new operating system, but using a compiler that is already supported. As with the new operating system, you first have to make sure the makefiles and related header files have your new compiler configured for them.

First of all, create the Compiler-Related Files. You can do this by copying files of another compiler that is close to yours, and then modify them to fit your compiler. You can find these files in the *common/config/tool* and *common/config/make* directories. Then do the following:

1. When writing the makefile for your compiler (*common/config/make/tool_<compiler>.mak*), do not forget to set the LEGAL_COMPILER_OS to the list of operating systems supported by this compiler and LEGAL_COMPILER_TOOL_VERSION to the versions of the compiler you want to support.

   Two additional parts should be taken care of gently in this file:

   - The compiler options to use. Go through them carefully and set them to fit your compiler of choice.
   - The dependency file generation. This is used by the MAKEDEPEND variable at the end of the makefile you are modifying. This variable holds the parameters used to generate a *.d* file for each source file of the project.
2. When dealing with the header file for the compiler, go over the list of general types declared, to make sure all of them are set correctly.
3. Add the new compiler to the list of supported compilers. This list can be found in *common/config/tool/rvtooldefs.h* Do not forget to define the specific compiler versions you intend to support.
4. Add this definition and the name as a string to the *common/ccore/rvccorestrings.h* file. This string can be fetched from within the application by calling the RvCCoreInterfaces() function to identify the configured compiler that is used and to solve troubleshooting problems in the future.
5. Add an include directive to your newly created header file to *common/config/tool/rvtoolconfig.h*.

6. In the file *common/config/make/environment.mak*, add the new compiler to the list of supported compilers specified by the variable LEGAL_COMPILER_TOOLS.
7. Modify the *default.mak* file with the information of your new compiler. Don't forget to set the version as well as the compiler itself in this file.

You are now ready to go through the Low Level modules described in this document and port each one of them appropriately. To some of these modules you will probably find interfaces already implemented that will fit your needs, while other modules will need to have new interfaces that you implement.

Adding a New Interface to a Module

If, during the porting process, you find a module that needs a new interface to fit to your operating system, you will probably have to add that interface into the code.

The following example illustrates how to add a new interface to the Timestamp module.

1. Give a name to your new interface and add it to the list of interfaces for that module. The list of interfaces for all modules is in *common/config/rvinterfacesdefs.h*. In this case, the interface is called BRANDNEW. This will add the definition RV_TIMESTAMP_BRANDNEW as the last one in the interfaces of the Timestamp in this file.
2. Add this definition and the name as a string to the *common/ccore/rvccorestrings.h* file. This string can be fetched from within the application by calling RvCCoreInterfaces() function in order to identify the selected configuration and to solve trouble-shooting problems in the future. To the list of Timestamp string names you will probably add something such as:

```
#elif (RV_TIMESTAMP_TYPE == RV_TIMESTAMP_BRANDNEW)
#define RV_TIMESTAMP_STRING "BrandNew"
```

3. Add the new definition of the interface as a valid interface in the *.h* file of the module you are porting. This will add the (RV_TIMESTAMP_TYPE != RV_TIMESTAMP_BRANDNEW) to the list of sanity checks in the beginning of *common/ccore/rvtimestamp.h*.
4. Now implement the functionality of the BRANDNEW timestamp. To do this, use one of the following methods:
   - The first method involves putting the new code directly into each of the timestamp functions, framed by #ifdef / #endif statements.
   - The second method can be used only in modules that have been implemented using the Adapters concept. You can determine whether the module you are porting supports Adapters by searching for function names starting with RvAdXXX that are called by functions of the module. According to the Adapters concept, the module contains only the generic part of the interface implementation and does not need to be modified.

Instead, put your new code in a new C file in a separate folder located in the *common/adapters* directory. Then add the name of the new folder to the Adapter folder list in the make file appropriate to your operating system (*common/config/make/os_xxx.mak*). The Adapter list is specified in the make variable ADAPTERS_$(TARGET_OS). The *common/adapters* directory contains many examples of Adapters for you to examine.

Example

In the timestamp module (*common/ccore/rvtimestamp.c*), find the function *RvTimestampGet()*. You will see that the function calls *RvAdTimestampGet()*, which is implemented in several Adapters according to the required operating system.
For example, the Adapter for Windows is located in *common/adapters/windows*. This folder contains the implementation of "getting timestamp" under Windows.

INDEX

S

T

U